Performance Analysis of Scheduling Policies for a Class of Flexible Automated Manufacturing Systems through Simulation

by

Christopher Nectarios Peters

B.S. in Electrical Engineering, Rutgers University, 1994

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1996

Author .....
Department of Electrical Engineering and Computer Science
January 30, 1996

Certified by .....
Dimitri Bertsekas
Professor of Electrical Engineering
Thesis Supervisor

Certified by .....
John Tsitsiklis
Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Committee on Graduate Students

Performance Analysis of Scheduling Policies for a Class of Flexible Automated Manufacturing Systems through Simulation

by

Christopher Nectarios Peters

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

Abstract

A class of flexible manufacturing systems general enough to embrace many of the complexities of both present and future discrete part manufacturing systems is proposed and modeled. Products move through a typical system on pallets via an Automated Guideway Vehicle System (AGVS). A distributed cooperative control view is taken towards modeling. Systems within the proposed class are described as the aggregate of many self contained intelligent system-level components. The behavior of these system-level components is also modeled, as well as a communications architecture required to coordinate their decisions. A tool for simulating members of the class is implemented in C++. An object-oriented approach is taken in developing the simulation tool so as to capture the open system characteristics inherent in distributed cooperatively controlled manufacturing systems. The tool is then used to simulate a simple system under the control of two different scheduling policies. Successful simulation will verify the correctness of the system model. Performance comparisons of the two scheduling policies are made. The results of the comparison will justify the use of the simulation tool as a mechanism for testing scheduling policies on members of the proposed class of systems.

Thesis Supervisor: Dimitri Bertsekas
Title: Professor of Electrical Engineering

Thesis Supervisor: John Tsitsiklis
Title: Professor of Electrical Engineering

Acknowledgments

I would like to express special thanks to: My parents, Stratos and Vasiliki, for raising me with love and support; Leanne Attai, for her love, support, and encouragement; Professor Michael Athans for giving this thesis its initial direction; Patrick Kreidl for his input and collaboration; Professor Dimitri Bertsekas and Professor John Tsitsiklis for acting as my advisors; and Square D, Groupe Schneider for partially funding this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Manufacturing Background

In recent years, due to factors such as decreasing product life cycles and the rising cost of inventory space, there has been a trend in the manufacturing community towards a manufacturing-to-order philosophy. The manufacturing-to-order philosophy emphasizes the filling of customer orders by manufacturing the required products as customer orders arrive. Also, due to increased global competition, manufacturers have been trying to lower overall costs and raise product marketability. One way of increasing marketability is by offering a wide variety of similar products, or product families.

To accommodate these trends, companies desire an industrial manufacturing system with both product and process flexibility [1]. Product flexibility refers to the manufacturing system's ability to manufacture more than one product type. Typical Flexible Manufacturing Systems (FMSs) have this attribute. FMSs are characterized by collections of flexible workstations. Flexible workstations, each capable of performing multiple tasks, are interconnected via an automated part handling system such as conveyer belts or guideways. Furthermore, if a task can be performed by more than one flexible workstation, then the FMS may exhibit process flexibility. This means that for a product being processed, there is more than one possible sequence of workstations that it can be routed through.

A major challenge towards making manufacturing systems more flexible is overcoming the associated increase in complexity. Technological advances in communications and computer hardware and software have sustained the growth of complex manufacturing systems. However, as the complexity of manufacturing systems has increased, so has the difficulty of

controlling them. Automated manufacturing systems require scheduling policies to control their production. For systems with product flexibility, scheduling policies must dictate, at any given time, which products to initiate production of. In systems exhibiting process flexibility, scheduling policies are also used to decide which workstation products should be routed to. For most practical automated manufacturing systems, it is not possible to derive optimal scheduling policies analytically. Clearly, a method for developing good scheduling policies for complex manufacturing systems is needed. One way of accomplishing this is through the use of computer

simulation.

1.2 Thesis Motivation

Given the problem described in section 1.1, this thesis has been motivated by the concept of employing distributed cooperative control in manufacturing systems. The distributed cooperative view of control systems recognizes that very complex systems, such as modern manufacturing systems, are beyond direct centralized control. A distributed cooperatively controlled manufacturing system may be defined by specifying the behavior within its self-contained intelligent system-level components, as well as the communication architecture these system-level components require to coordinate decisions.

The general setting for studying this problem is based on a proposed class of manufacturing systems, hereafter referred to as a Multi-Product/Multi-Stream (MPMS) system. An MPMS system is general enough to embrace the complexities associated with both present and future factories. The proposed MPMS system model supports the study of a wide variety of candidate distributed and cooperative control architectures [3]. This thesis focuses on the software modeling and implementation issues related to simulation of the proposed MPMS system, with the intent of studying scheduling policies. For this reason, the same distributed cooperative control architecture will be used regardless of the system and/or scheduling policy under consideration.

In the context of distributed cooperatively controlled manufacturing systems, a candidate MPMS configuration, at the system-level, is described as the aggregate of a number of system-level components. These components will be controlled via a distributed cooperative control architecture. At the local level, this architecture will support the behavior

of system-level components towards pallets, the mode of transportation for products being processed through the system. The architecture also supports the coordinated transfer of pallets between neighboring system-level components at the local level. At a global level, the control architecture will allow for the communications necessary to route pallets and implement scheduling policies.

1.3 Thesis Overview

An object-oriented approach will be taken towards modeling the proposed MPMS system and developing the simulation tool [2], [7]. Using this approach, the MPMS will be conceptually broken down into its fundamental system-level components. Next, these components will be defined by specifying their internal structure and their behavior when interacting with other objects. Finally, the MPMS system model will be built up using the system-level components. In order to simulate the behavior of a candidate MPMS system under various scheduling policies, orders with random compositions and arrival times will be defined. The scheduling policies will focus on the manufacturing-to-order concept, since it is suitable when instances of the class exhibit both product and process flexibility.

The fundamental attributes of the object-oriented paradigm seem well-suited for achieving the software modeling and implementation goals mentioned in the architectural considerations. Data encapsulation and the concept of polymorphic behavior are among the most noteworthy. Data encapsulation reinforces the idea of distinct system-level components whose internal structure may not be accessible to other objects. Interactions between encapsulated objects occur through the exchange of messages. Finally, an object is said to exhibit polymorphic behavior when its behavior to a request is specified only by the object's type and the request.

The simulation tool will be developed in C++, a popular programming language which supports the object-oriented paradigm [5]. In C++, data encapsulation is implemented via the definition of classes, of which objects are specific instances. Each class contains a set of attributes and member functions. The accessibility of the contents of a class to other classes may be specified by the programmer. Furthermore, C++ facilitates the implementation of polymorphic behavior through inheritance and virtual functions.

1.3.1 Thesis Objectives

The overall goal of this thesis is to show the applicability of using computer simulation to obtain insight for developing good scheduling policies for the MPMS class of manufacturing systems. There are three main objectives. First, this work will aim to introduce and define the MPMS class of manufacturing systems and its capabilities. Once the class has been defined, the next objective will be to implement its model using the object-oriented programming language, C++, so that quick construction of manufacturing system models within the general topology is possible. The implemented model will be able to simulate manufacturing systems under different scheduling policies in order to select scheduling policies which lead to higher system performance. The third objective of this work is to construct a simple system within the general topology and successfully simulate its performance under two different scheduling policies. The comparative performance of these two scheduling policies will then be

shown to conform to expectation. Successful simulation of the manufacturing system will also verify the correctness (the model behaves as expected and is stable) of the implemented manufacturing system model.

1.4 Thesis Organization

Chapter 2 will provide a description of the proposed MPMS Flexible Manufacturing System class. It will also discuss the MPMS system-level components and their functions. In Chapter 3, an in-depth discussion on the object-oriented C++ system implementation will be provided. The simulation experiment discussed in the thesis objectives will be presented in Chapter 4, along with the results of several simulation runs of each scheduling policy. Finally, Chapter 5 will conclude the thesis with a review of the work presented and comments on the success of the simulation.

Chapter 2

Proposed Class of Automated Manufacturing Systems

2.1 Restrictions and Assumptions

The class of automated manufacturing systems being considered deals only with discrete-part manufacturing, where parts travel through a collection of flexible workstations via a pallet-based transfer system, such as an Automated Guideway Vehicle System (AGVS). All pallets are assumed to be identical irrespective of the type of part they carry. In this thesis, it is assumed that all parts to be manufactured belong to a common part family. Furthermore, it is also assumed that throughout the manufacturing process, a part will be processed on exactly one pallet. That is, scenarios such as those in which two pallets carrying part subassemblies meet at a workstation and are combined into an aggregate part are not allowed.

Some basic assumptions and restrictions are also made about orders and their arrival. First, orders to be manufactured are of random size and composition. Also, order arrivals will be stochastic. In Chapter 4, the assumption is made that order interarrival times follow a Poisson distribution and that their size and composition are uniformly distributed. Two restrictions are placed on the way orders are manufactured, in accordance with the manufacturing-to-order paradigm. First, processing of an order begins only after it is placed, therefore, no part inventories are kept in anticipation of future orders. Upon arrival, customer orders are entered into a queue until processing can begin. Second, all finished parts

n	number of layers in the system
L_i	ith layer controller in the system
I_k	kth input stream of the input layer
O_k	kth output stream of the output layer
W_{ik}	kth flexible workstation of central layer i
M_{ik}	kth flexible machine of central layer i
B_{ik}	kth buffer of central layer i
H_{ik}	kth guideway link of layer i
J_{ik}	kth guideway junction of central layer i

Table 2.1: Figure 2-1 Key

pertaining to a unique customer order must be collected at the same output stream.

2.2 General Overview of Topology

Figure 2.2 shows the topology of the proposed system class. The fundamental system-level components depicted in the figure are listed in Table 2.1. The proposed class is characterized as a Multi-Product/Multiple Stream (MPMS) FMS due to the fact that its instances may have the attributes of both product and process flexibility. Product flexibility arises through the choice of flexible workstations so that the system can produce more than one part type. Process flexibility is due to multiple streams which may allow pallets a choice of workstations for a given operation. The proposed class is of sufficient generality so that simpler line configurations such as Single-Product/Single-Stream (SPSS), Single-Product/Multi- Stream (SPMS), and Multi-Product/Single-Stream (MPSS), may be viewed as special cases.

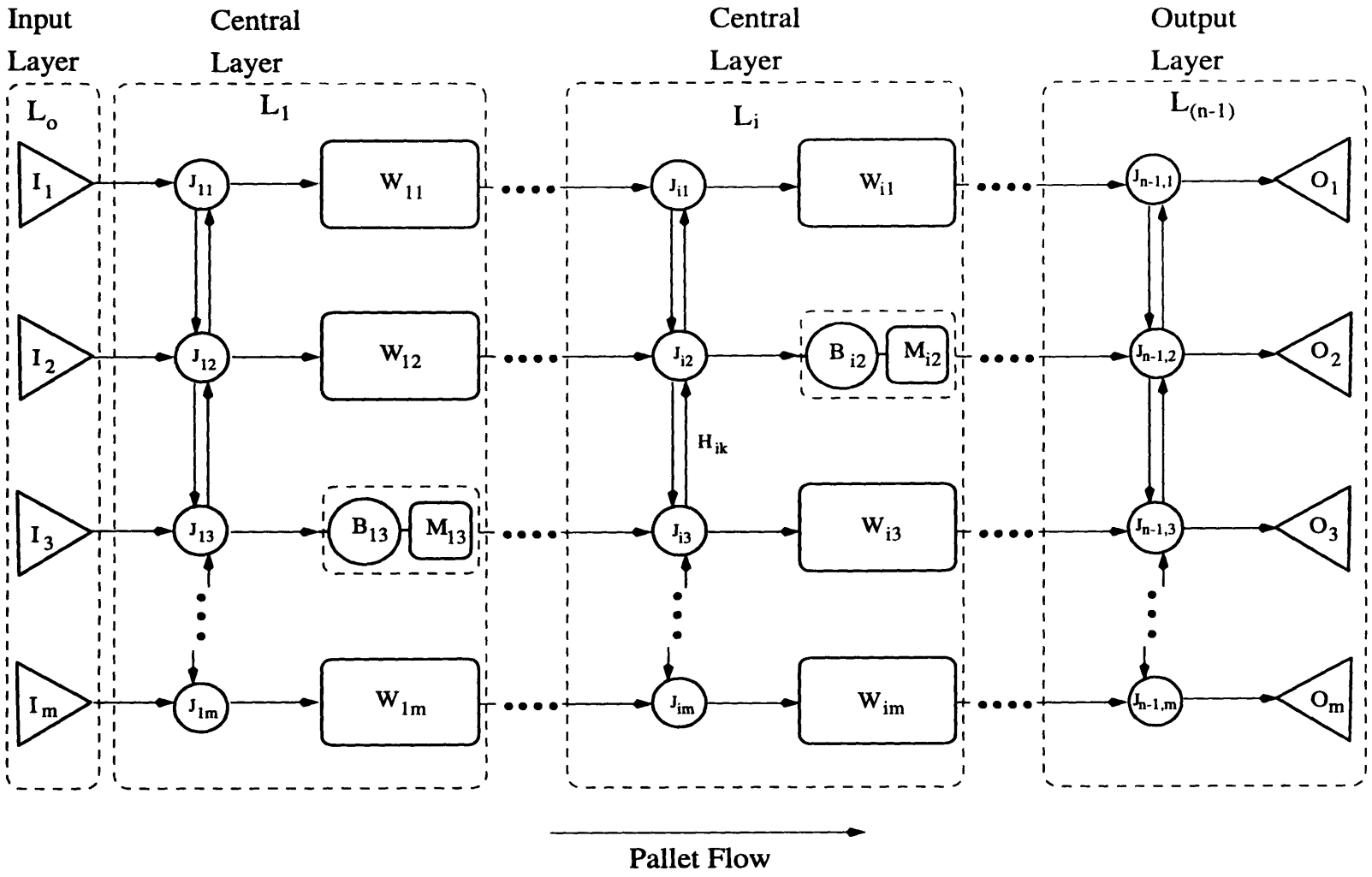

Figure 2-1: MPMS System

In order to incorporate the concept of product-based layout, an MPMS line configuration is designed to manufacture a single product family. The layout of an automated manufacturing system within the proposed class is specified by considering the sequence of tasks required by all members of its associated product family. Flexible workstations are grouped into layers, based on the set of tasks they perform, and at what stage within the part type processes these tasks fall. Proper grouping ensures that under nominal circumstances, pallets traversing the system are assured access to workstations capable of handling the tasks they require.

All pallets carrying "raw" parts enter the system via input streams in the input layer. Note that pallets belonging to the same order are not necessarily restricted to enter the MPMS system from the same input stream. Upon entering each central layer, a pallet is routed to a workstation, undergoes its required task, and moves to the next layer. A pallet visits only one workstation in each central layer. As shown in Figure 2.2, pallets flow through the layers from left to right, so once a pallet leaves a layer it never enters that layer again. Consequently, if a pallet is carrying a part which needs the same task to be performed multiple times in the course of processing, workstations capable of performing that task must exist in multiple layers. By the time a pallet enters the output layer, the product it is carrying is complete. In the output layer, the pallet is routed to its appropriate output stream.

2.3 System-level Components

At any given time, the state of the system is given by the customer orders in the order queue, as well as the distribution and type of pallets within the system. Here, pallet type refers to the product type the pallet carries since all pallets are identical when not carrying products. Including the completed tasks on a pallet's unfinished product in the state space would be redundant because that information is known from the location of the pallet in the system. However, it is important that pallets undergoing tasks at a workstation also be considered when determining the state of the system. The amount of time which the pallet has been at the workstation determines how much longer it will be until the workstation is available to perform a task on another pallet. This information is essential to routing other pallets. The state of the system, minus the order queue, can then be seen as a union of the

distribution and type of pallets within every system-level component, constrained by the way these components are interconnected to form the system.

There are five types of system-level components which may be used to construct a manufacturing system within the proposed class: input stream, link, junction, workstation and output stream. The remainder of this section gives a description of the five system-level components, leaving issues of system control for the final section of this chapter.

Input streams are the sole components of the input layer. Each input stream can hold at most one pallet. In order for a pallet to be loaded onto an input stream, there must exist a path from the input stream to a sequence of workstations capable of processing the pallet. If an input stream has a pallet, it attempts to move the pallet to its downstream link, provided that prespecified discharge conditions are satisfied. Conditions for a pallet to be discharged from input streams may, for example, be used to control pallet congestion in downstream layers. Once the discharge conditions are satisfied, the input stream is responsible for the successful loading of its pallet onto its downstream link. The input stream can sense whether its downstream link is backlogged with pallets.

The guideway system used to facilitate the movement of pallets within the system is made up of links and junctions. Although not apparent in Figure 2.2, links may be of varying lengths in order to incorporate the varying spatial distribution between system-level components such as the distance between two workstations in different layers. The length of a link is measured by the maximum number of pallets it can hold if the pallets are lined up end to end. Pallets moving unobstructed are assumed to travel at the same speed on every link. Links are responsible for successfully transfering their pallets to their downstream system-level components. In order to do this, links can sense when their downstream components are ready to accept a pallet. If a pallet is on a link and it is blocked, the pallets on the link behind it will continue moving down the link until they too are blocked.

A junction is found wherever multiple links intersect. The primary function of a junction is to facilitate the transfer of pallets from one link to another. When a pallet reaches the end of a link, the link's downstream junction queries the pallet as to which of the junction's downstream links it has been assigned to visit. The junction then senses whether or not the assigned link is ready to accept a pallet. If a pallet can be accepted, then the junction passes the pallet. Otherwise, the pallet stays at the end of the link, blocking it. In the course of querying a pallet, the junction may find that a pallet requires assignments to downstream

links. If this happens, the junction is then used to communicate with the layer controllers to get a set of link assignments. The junction then attempts to route the pallet to its newly assigned link of destination.

Workstations, found in the central layer, perform the tasks necessary for the parts being carried by pallets to be completed. Workstations may or may not have buffers. If a workstation has a buffer, pallets entering the workstation are placed directly into the buffer. The workstation machine is given a pallet for processing based on the prespecified discipline of the buffer. If the workstation has no buffer, then a pallet entering the domain of the workstation is placed directly into the machine's processing station. When pallets enter the machine's processing station, the machine recognizes the product type and performs the task which is required for that product. In order to accomplish this, each workstation machine is preprogrammed with a mapping telling it what tasks to perform for any given part. After the machine has completed the task, it then attempts to pass the pallet to its downstream link.

Output streams, found in the output layer, serve as "collection points" for pallets. When a pallet enters an output stream, the pallet's arrival is recorded in the output stream's data base. The finished part is then removed from the pallet and it awaits the arrival of the remaining parts from its associated order at the output stream. Once all the necessary pallets arrive, the order is shipped. To simplify the restriction that all parts of an order must be received by the same output stream, it is assumed that any output stream may be reached by any workstation in the central layer immediately preceding the output layer.

2.4 System Control and Coordination

In addition to carrying an unfinished product, a pallet also possesses the information necessary for routing itself through the system. First, a pallet carries identification with information that includes the type of part it carries, what order it belongs to, when that order was issued, when it is due, and which output stream it is heading for. It also has onboard memory for storing a sequence of link assignments and a sequence of workstation assignments. Depending on the type of scheduling system the MPMS system has, a pallet may have one or more workstation assignments at any given time. However, the sequence of link assignments will only be in regards to the workstation the pallet is to visit next.

There are three types of layers in which the system-level components are grouped. They are the input, central, and output layers. The notion of layers arises as a convenient way of organizing the MPMS system. A layer is more than just a physical grouping of objects, it is also a region where information about system-level components and their pallets are directly available to a layer-controller. Every layer may be viewed as a region controlled by exactly one layer-controller. Each layer-controller has direct access to all information regarding the components within its layer. Based on the information it gathers, the layer-controller can direct the flow of pallets through its system-level components. A layer-controller handles all decisions which are not automatically handled by its system-level components. There may be cases in which a layer-controller requires information about system-level components which are not in its layer in order to make decisions. In these cases, a layer-controller must obtain the information it requires from the layer-controller of the relevant system-level components via a communications network for layer-controller to layer-controller communications.

The input layer-controller is in charge of loading "raw" pallets onto the input streams. First, based on a preprogrammed selection criteria, the input layer chooses a customer order to begin processing. Next, the input layer-controller selects a part from that order and then determines which input stream to load the part's pallet onto. In making these decisions, the input layer-controller must consider several things. One thing which needs to be checked, at a minimum, is the input streams, to see which ones are available and which parts of a product family they can accept. Another factor which the input layer-controller must consider is the order queue and which parts still need to be made. Finally, the input layer-controller should also consider customer order size and priority, and guideway traffic.

A central layer-controller is responsible for routing pallets to its workstations, and hence onto the next layer. A central layer-controller has, at its disposal, direct access to information from every system-level component within the central layer. It uses this information, along with information from outside the layer (obtained through the network) to make routing decisions. Pallets obtain instructions to workstations by communicating to central layer-controllers at junctions. Communication with a central layer-controller is triggered if upon reaching a junction, a pallet does not have a next link or workstation assignment. If a pallet already has a workstation assignment, then the layer-controller gives the pallet the sequence of links it must pass through in order to reach the workstation. If a pallet requires

a workstation, then the layer-controller assigns the pallet a workstation to visit, and gives the pallet the sequence of links that it must pass through in order to reach the workstation.

The output layer-controller is responsible for assigning customer orders to an output stream when they are placed. It is also responsible for giving pallets the sequences of links they must pass through in order to reach their assigned output streams. Junctions in the output layer sense whether or not pallets need directions to their assigned output streams. If a junction finds a pallet which needs directions, then the junction informs the output layer-controller, which then gives the pallet the necessary sequence of links to pass through.

Chapter 3

Object-Oriented Implementation

3.1 Object-Oriented Model Implementation using C++

The MPMS system simulator was programmed in C++. C++ is among the most popular object-oriented programming languages in use today. C++ was chosen over procedural languages such as Pascal, C, and BASIC due to its support for encapsulation, inheritance, and polymorphism. Encapsulation is comprised of two subordinate concepts: bundling and information hiding [5]. Bundling associates a set of functions with a data structure as the only functions allowed to operate on that data structure. C++ implements bundling by assigning class membership to functions. Classes are the means by which abstract data types are implemented. Information hiding limits access by clients to data structure members and bundled functions. C++ provides class access qualifiers private and public as the mechanism for implementing information hiding. Inheritance, more specifically public inheritance, is the C++ mechanism for implementing generalization and specialization relationships among classes. Generalization captures similarities between different abstract classes such as common characteristics, associations, and functionality. Specialization captures differences between similar classes. An object is said to exhibit polymorphic behavior when its behavior to a requested operation is specified by the object's class, the requested operation, and no other criteria. C++ facilitates the implementation of polymorphic behavior through inheritance from abstract base classes and the use of virtual functions. The concepts of encapsulation, inheritance, and polymorphism will be made clear and concrete in this chapter.

This chapter provides a detailed description of how the C++ program was used to

simulate the MPMS system. The discussion will try to minimize the usage of C++ in describing the simulation, but will define the objects used, their attributes and operations, and the interactions between them. First, the method of generating customer orders and the modeling of the queue will be described. Next, a brief section on the interaction of the system-level components will set the stage for a discussion on the modeling of each of the five system-level components; the input stream, link, junction, workstation, and output stream. Finally, a description of the system layer-controllers; input, central, and output will conclude the chapter. For detail on the C++ code, a copy of the C++ implementation of the MPMS system model developed in this thesis is given in Appendix B.

3.2 Order Generation

In generating orders, we must first decide how many different types of products a product family will contain. This is also of fundamental importance for designing the MPMS system capable of producing the products. We must now make assumptions in reference to the size of orders, the distribution of part types within orders, and the frequency of order arrivals. In Chapter 4, we will assume that the size of orders is random and uniformly distributed between 1 and some maximum order size. Furthermore, we will assume that the composition of each order is random and that each part of an order has an equal probability of being any part in the product family. C source code for generating random orders from a product family with n members having the random distribution just described may be found in Appendix A.

In order to implement the MPMS simulator, time will be discretized into units which will be referred to as seconds. All system operations, pallet movements and customer order arrivals take place or occur at integral multiples of one second. For example, in one second we assume that a pallet travels one pallet length down a link. Also, in Chapter 4 we assume that incoming customer orders are queued once a minute. Figure 3-1 shows the object diagram for the abstract data type Time, one of whose instances will act as the system timing mechanism by which all system activity and stimuli are orchestrated. Object diagrams contain the name of the abstract data type in the top third. The names of attributes and abstract data types in the middle third and operations in the bottom third.

Time's attributes are day, hour, minute, and second. Time functions just like a digital

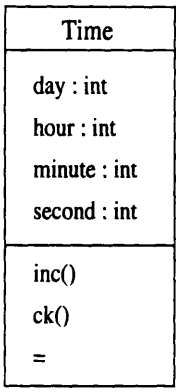

Figure 3-1: Object Diagram for Abstract Data Type Time

stopwatch set on military time. Time::second is incremented by 1 using the Time::inc() operation. The Time::ck() operation is used by clients who wish to obtain a Time object's values for day, hour, minute, and second attributes. C++ allows for the overloading of built-in operators. The implementation of class Time (classes are all named after their respective abstract data type) required the overloading of the binary addition operator, +, and the assignment operator, =. These overloaded operators allow for the quick addition of two Time operands in the natural sense. For example, if a, b, and c are instances of Time, then c = a + b sets the time c equal to the sum of the representative times in a and b. Note that overloaded binary addition operator was implemented as a non-member friend of class Time with global scope.

The arrival of customer orders is assumed to be a Poisson Process. A Poisson Process is a continuous time, discrete amplitude random process [8]. The definition of a Poisson Process is as follows:

Definition: Let $G(t)$ be an integer-valued random process. $G(t)$ is said to be a Poisson Process if the following conditions hold:

a. For any times $t_1, t_2 \in \Gamma$ and $t_2 > t_1$, the number of events $G(t_2) - G(t_1)$ that occur in the interval from t_1 to t_2 is Poisson distributed according to the following distribution Law.

$$P[G(t_2) - G(t_1) = k] = \frac{(\lambda(t_2 - t_1))^k}{k!} e^{-\lambda(t_2 - t_1)}, k = 0, 1, 2..$$

b. The number of events that occur in any interval of time is independent of the

number that occur in other nonoverlapping time intervals

From its definition, it can be shown that both the mean and variance of $G(t_2) - G(t_1)$ are equal to $\lambda(t_2 - t_1)$. A description of a method for generating Poisson deviates, as well as its implementation may be found in the book *Numerical Recipes in C* [6].

Every minute, using the system timing mechanism as a reference, Poisson deviates will be generated. For each deviate, the random order generator described above will generate the size and composition information required for a customer order. This information will then be used in the constructor argument of a class of orders which will be stored in the order queue. The implementation of the order and order queue abstract data types, which will be referred to as InOrder and InQueue, is discussed in the following section.

3.3 Queue Modeling

Figure 3-2 shows the object diagrams for the InQueue, InOrder, and OUT abstract data types, along with their associations to one another. OUT refers to the output stream system-level component. The association between types InQueue and InOrder is one-to-many and unidirectional. The three circles notation in the InQueue attribute section, with one circle anchoring an arrow, specifies that InQueue can contain an arbitrary number of InOrder pointers. There is a one-to-one unidirectional association between types InOrder and OUT. This association exists because each product in a customer order must be routed to the same output stream, hence the customer order is assigned to one output stream. The object diagram of OUT as shown in Figure 3-2 is incomplete because its attributes and operations are irrelevant in the context of this section's discussion.

Since only one InQueue object is required for the MPMS simulator, this object will itself be referred to as the InQueue. The InQueue is implemented as a doubly linked list of pointers to InOrder objects. InOrder objects registered in InQueue are linked chronologically in the order of their creation. InQueue has one attribute called order_id which is initialized to 1 at the beginning of a simulation. When an order arrives at InQueue, the InOrder object created using the InQueue::insert() operation is assigned the current value of InQueue::order_id, then InQueue::order_id is incremented by 1 in anticipation of the next arrival. An InOrder object is accessed by clients through the InQueue using its InOrder::order_id number. The function InQueue::isIn() takes an integer argument and returns the pointer to an InOrder object whose

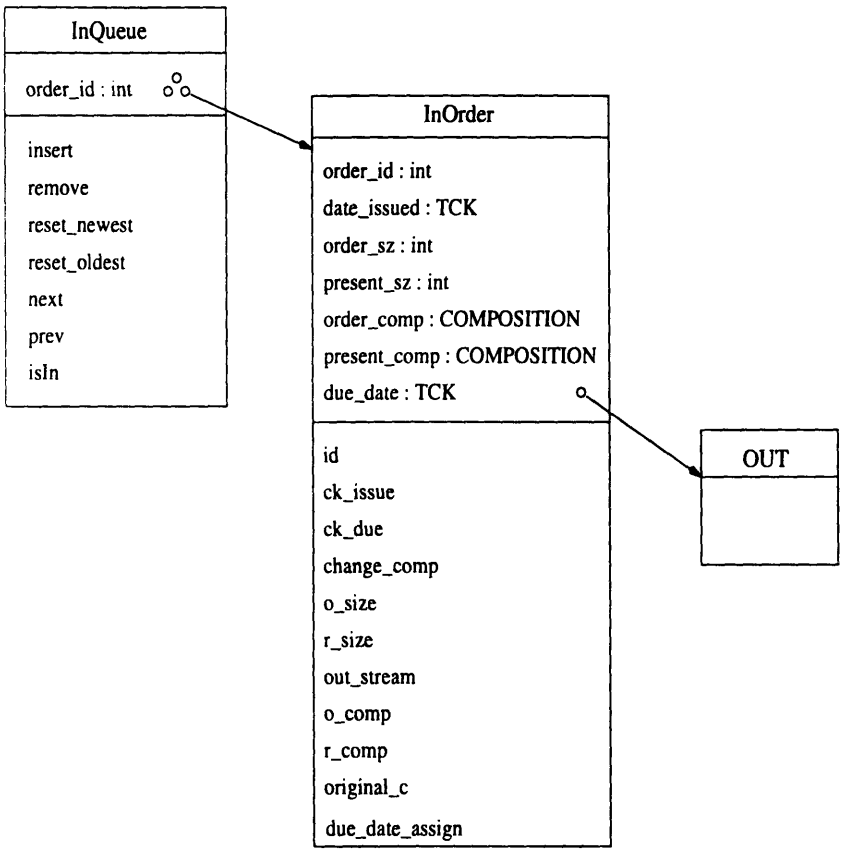

Figure 3-2: Object Diagram for Abstract Data Types InQueue, InOrder, and OUT

order_id is equal to the argument. Once a client has the pointer to the desired InOrder object, it can access the object's attributes using its operations.

When an InOrder object is created, it is assigned an order identification number, an output stream and a due date. The definitions of the attributes and operations of class InOrder are given in Tables 3.1 and 3.2, respectively. TCK is a data structure defined as follows:

```
struct TCK {int day; int hour; int minute; int second;};
```

COMPOSITION is a class which was defined so as to make the manipulation of the composition of customer orders easier. If a product family has n members in it, with the products named product 0, 1, 2, up to n-1, then a COMPOSITION object in an InOrder object is a vector of length n whose ith element is equal to the number of product i in the COMPOSITION object. The operations of class COMPOSITION are never used directly, rather they are used by the operations of class InOrder.

Attribute Name	Value Type	Description
order_id	int	Order identification number
date_issued	TCK	Date InOrder object was created
order_sz	int	Number of products in customer order
present_sz	int	Number of products which have not yet been initiated
order_comp	COMPOSITION	Distribution of customer orders
present_comp	COMPOSITION	Distribution of customer order which has not been initiated
due_date	TCK	Time which customer order must be complete
out	*OUT	Output stream customer order is assigned to

Table 3.1: Attributes of Class InOrder

Operation	Description
int id()	Returns order's id number
TCK ck_issue()	Returns date_issued
TCK ck_due()	Returns due_date
int o_size()	Returns order_sz
int r_size()	Returns present_sz
int o_comp(int PART)	Returns order_comp[PART]
int r_comp(int PART)	Returns present_comp[PART]
int change_comp(int PART, int CHANGE)	Changes the number of PART in customer order by CHANGE
COMPOSITION original_c()	Returns order_comp
void out_stream(OUT*)	Sets the output destination of the customer order
OUT* out_stream()	Returns the output destination of the customer order
TCK due_date_assign()	Sets the order's due_date, and returns due_date

Table 3.2: Operations of Class InOrder

In summary, every minute, using the system timing mechanism as a reference, Poisson deviates will be generated. For each deviate, the random order generator described previously will generate the size and composition information required for a customer order. This information will then be used in the argument of InOrder::insert() to create a new InOrder object. The new InOrder object will be assigned a unique order identification number. It will also be assigned a due date by some programmer specified rule. Throughout its lifetime, the InOrder object will also keep track of how many, and which of its products were initiated. Finally, when all of its products have been successfully routed, the InOrder object will be removed from the InQueue and deleted using the InOrder::remove() operation.

3.4 MPMS System Modeling Goals

The primary goal is to implement an MPMS system simulator which captures the qualities inherent in what has been termed in recent computer science literature as open systems. Open systems have the characteristics of concurrency; asynchrony; decentralized distributed control, based on possibly inconsistent information; and high dynamic adaptability requirements resulting from unrestricted joining and leaving subsystems [4]. The attributes of concurrency and asynchrony come into play by designing the simulator so that in any time interval, multiple system-level components may be performing operations, independently of one another. Control for the MPMS system will be decentralized and distributed. The system-level components will be implemented as intelligent subsystems capable of performing many of their operations independent from any centralized controller. The characteristic of high dynamic adaptability is particularly important in the system simulator. The simulator is supposed to be capable of simulating any system within the class of MPMS systems. This will be accomplished by building a system within the proposed class piece by piece from system-level components and specifying the system and subsystem level decision rules that control the system. It is desirable that once a system and its decision rules are specified, system-level components may be added or subtracted with very little system reprogramming and little or no modification to the decision rules.

3.5 Modeling System-Level Component Interactions

Successful development of the MPMS system simulator will allow for creation of systems piece by piece from system-level components. From the description of the class of MPMS systems given in Chapter 2, as well as Figure 2.2, it is clear that workstations and junctions will always have links as their associated upstream and downstream system-level components. It is also clear that an input stream is always directly upstream of a link, and an output stream is always directly downstream of a link. Therefore, a link may have several possible combinations of upstream and downstream system-level components which it must interact with. If the behavior of system-level components interacting with one another is defined to be polymorphic, this would not be a problem. Define COMP as the abstract generalization whose concrete specializations are the five system-level components; input stream, link, junction, workstation, and output stream. COMP's operations are the only operations used by the system-level components to interact with one another. When a system-level component requests an operation from one of its neighbor components, it does so as if it was requesting the operation from an instance (object) of COMP. Of course, no direct instances of COMP exist, since no direct instances of any abstract generalization exist. Therefore, the request is propagated to one of COMP's specialized types, namely the system-level component the operation was intended for.

Figure 3-3 shows the design model for the COMP abstract generalization hierarchy which was used to implement most of the system-level component interactions. The descendants of class COMP; IN, LNK, JCT, WST, and OUT are the C++ implementation of the input stream, link, junction, workstation, and output stream, respectively. The model explained in this section contains enough structure so as to be able to join system-level components together. The model will be completed in section 3.7 when the remaining attributes of the system-level component classes are included. C++ uses abstract base classes to implement abstract generalizations. The object model for abstract base class COMP is shown at the top of Figure 3-3. COMP has 1 attribute; comp_typ, and eight operations. Seven of the operations; set_up, set_dw, upstream, downstream, COMP_type, access, and move, are implemented as pure virtual functions. That is, there is no implementation for these functions in COMP. The descendants of COMP inherit only the interface for these functions which they must then implement themselves.

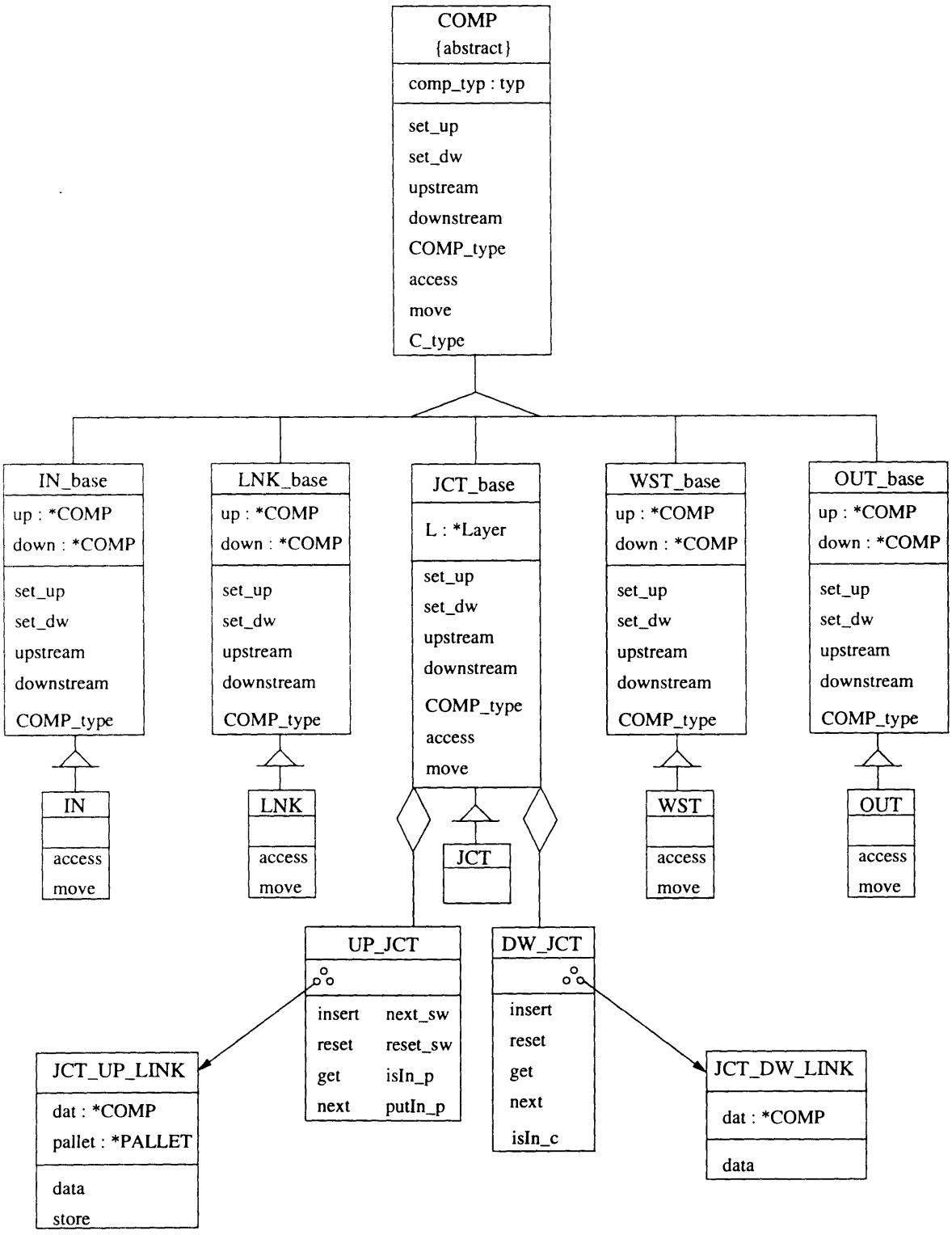

Figure 3-3: Abstract Data Type COMP Hierarchy

The behaviors that these virtual functions enable in the system-level component objects is described below:

virtual void set_up(COMP * U) Used to associate a system-level component object as the object directly upstream of another system-level component object. In order to set system-level component object A, as the object directly upstream of system-level component object B, the command B.set_up(&A) is used. Note that a JCT object may use this function multiple times, since it may have multiple upstream components, while a IN object never uses this function since it does not have any.

virtual void set_dw(COMP * D) Used to associate a system-level component object as the object directly downstream of another system-level component object. In order to set system-level component object B as the object directly downstream of system-level component object A, the command A. set_dw(&B) is used. Note that a JCT object may use this function multiple times, since it may have multiple downstream components, while a OUT object never uses this function since it does not have any.

virtual COMP * upstream() This function returns a pointer to the object directly upstream of the system-level component object from which it was called. The behavior that this function generates differs based on which object it was called from. The pointer returned from calls to a WST object's upstream function is always the pointer to the WST object's upstream component. The same statement can be made for calls to the upstream functions of LNK and OUT. However, the pointers returned from repeated use of a JCT object's upstream function are the pointers obtained by cycling through a list of pointers to the JCT object's upstream components. Also, when called from an IN object, a zero pointer is returned, since members of class IN have no upstream components.

virtual COMP * downstream() This function returns a pointer to the object directly downstream of the system-level component object from which it was called. The behavior that this function generates differs based on which object it was called from. The pointer returned from calls to a WST object's downstream function is always the pointer to the object's downstream component. The same statement can be made for calls to the downstream functions of LNK and IN. However, the pointers returned from repeated use of a JCT object's downstream function are the pointers obtained by

cycling through a list of pointers to the JCT object's downstream components. Also, when called from an OUT object, a zero pointer is returned, since members of class OUT have no downstream components.

virtual typ COMP_type(int L, int ID) This function is used to set the comp_typ attribute of the system-level component object it is called from. The first argument, L, specifies the layer the object will be in. The second argument, ID, is the unique identification number that the system-level component object will have in its associated layer.

virtual int access(COMP * U) This function returns a value signifying whether the system-level component object from which it was called is ready to accept a pallet from its associated upstream component with pointer U. A value of 1 signifies that the object is ready, i.e. the location that the pallet from the upstream component will occupy is currently vacant. A value of 0 implies that the object is not ready to accept the pallet. Note that although IN objects have no upstream components, their associated layer-controllers use access for the same purpose just described. Also, OUT objects are always accessible.

virtual void move(COMP * U, PALLET * P) This function passes the pallet pointer P from the system-level component object with pointer U to its downstream component from which the function was called. Note that in order for this function to be successful, the system-level component object must be ready to accept a pallet from its upstream component specified by U. Additional behavior for this virtual function will be given in the sections describing junction and output stream modeling.

The function join(COMP & U, COMP & D) has been defined as a friend to class COMP. It is used to associate the object U referenced by as the upstream component of the object D referenced by. The remaining COMP operation is named C_type, and is implemented as a nonoverriden nonvirtual function. The purpose of C_type is to return a structure typ which has the following definition:

```
struct typ {int layer; int object; int id_num;};
```

The structure typ is used by a client to identify a system-level component. The first variable, layer, is the layer that a system-level component resides in. object is the type of system-level

component. id_num is the unique identifier associated with a system-level component in the layer.

The direct descendants of COMP are the classes; IN_base, LNK_base, JCT_base, WST_base, and OUT_base. In these classes, the attributes and operations necessary for joining the system-level components are introduced. With the exception of JCT_base, all of their implementations are very similar. Each class has two attributes, up and down. They are used by the system-level component class instances that inherit them. up and down store respectively, pointers to their upstream and downstream COMPs. It follows that instances of IN always have up set to zero, and instances of OUT always have down set to zero. The implementation of class JCT_base is complicated by the fact that a junction may have an arbitrary number of upstream and downstream links. For all other direct descendants of COMP, it was possible to implement all virtual functions except for access and move without delving into the internal structure of their associated system-level component classes by simply using the two attributes, up and down. Defining the internal structure of a junction cannot be avoided for the implementation of all of the virtual functions with the exception of COMP_type.

As shown in Figure 3-4, each junction will be able to hold a number of pallets equivalent to the number of upstream links it has. The capability of the junctions to hold pallets is so that pallets may, if needed communicate with the layer-controller to receive routing instructions. This description of a junction is captured in class JCT_base. Referring to Figure 3-3 once again, JCT_base is an aggregate class, containing two aggregate components, UP_JCT and DW_JCT. Class UP_JCT objects are implemented as singly linked lists to JCT_UP_LINK objects. Each JCT_UP_LINK object has two attributes. The first, dat, contains a pointer to one of the junction's upstream COMP objects. The second, pallet, may hold a pointer to a pallet which came into the junction using the COMP object (link object) referenced by dat. The JCT_UP_LINK functions data and store are used by UP_JCT to access dat and pallet, respectively. Note that PALLET is the class implementation of pallet, and will be defined in the following section. One thing to note about the implementation of class UP_JCT is that it uses two iterators to access JCT_UP_LINK objects. One iterator is used in the implementation of the virtual functions. The other is used in order to implement precedence rules for deciding which pallet can traverse the link in any given time interval, should multiple pallets be at the same junction wishing to do so. The algorithm used to establish the precedence

rules will be given in section 3.7.3. Class DW_JCT objects are implemented as singly linked lists to JCT_DW_LINK objects. Each JCT_DW_LINK object has one attribute, dat, which contains a pointer to one of the junction's downstream COMP objects. The JCT_DW_LINK function data is used by DW_JCT to access dat. The operations of classes UP_JCT and DW_JCT are used only in the implementation of the operations of classes JCT_base and JCT.

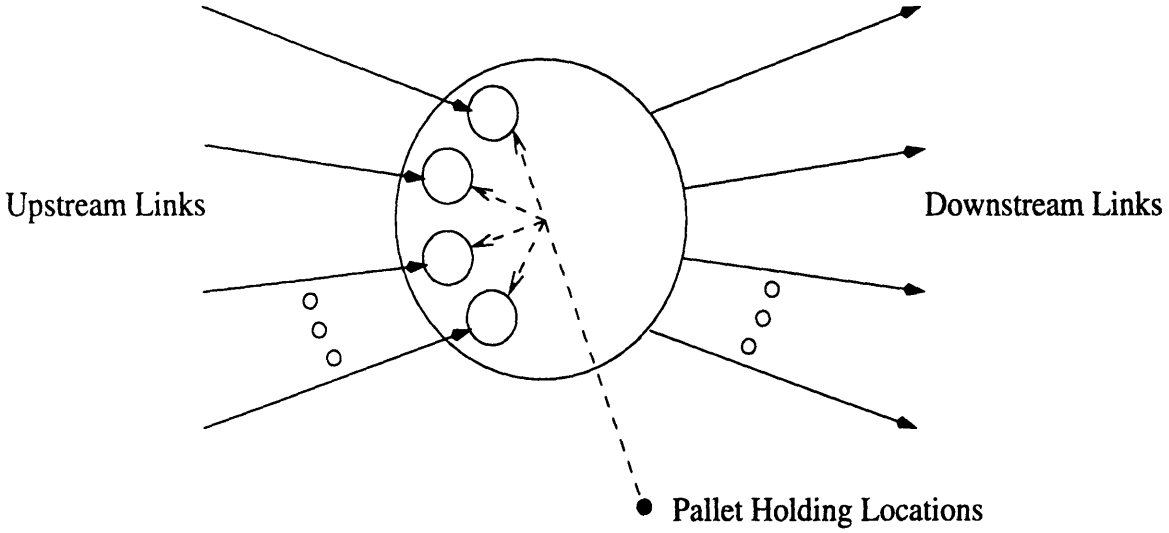

Figure 3-4: Junction Model

3.6 Pallet Model

Each pallet carries a part through the MPMS system so that it can have tasks performed on it by workstations. In the implementation of a pallet given in this thesis, a pallet carries more than just a part, it also carries information. Figure 3-5 shows the object diagram of abstract data type PALLET, which will be implemented by class PALLET. Tables 3.3 and 3.4 list respectively a PALLET object's attributes and operations, and what they are used for. When a pallet enters a central or output layer, it requires directions to get respectively to its destination workstation or output stream. The pallet is given directions by the layer-controller of the layer it is attempting to traverse. Direction is an array of pointers to LNK objects. The sequence of pointers in Direction matches the sequence of LNK objects a PALLET object must pass through to get to its next assigned WST object or OUT object starting from the first LNK object it encounters in the WST's or OUT's layer respectively.

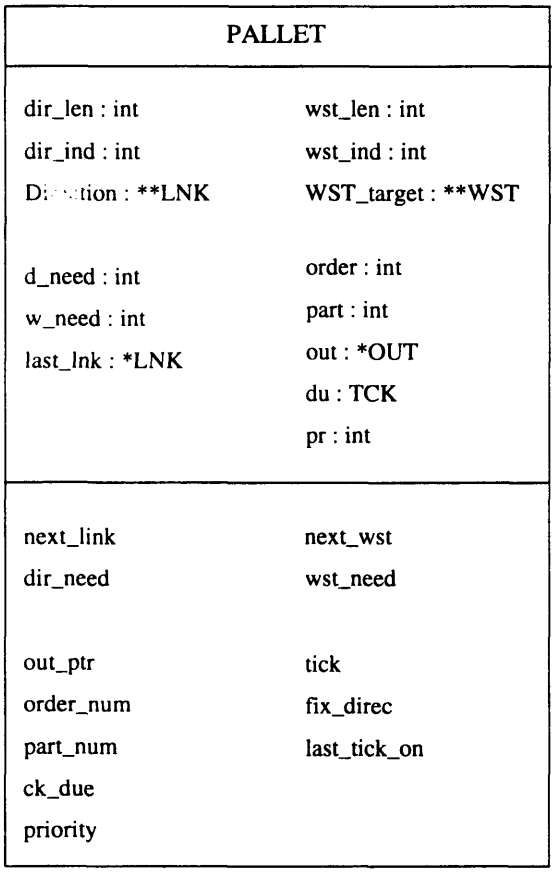

Figure 3-5: Object Diagram for Abstract Data Type PALLET

In order to receive directions to an assigned workstation, a pallet must first have an assigned workstation. Hence, a pallet also has the ability to store workstation assignments. Depending on the method by which pallets are controlled, at any given time a pallet may only have a pointer to its next workstation assignment, or it may have pointers to the next several workstations it must visit in the upcoming layers. In any case, WST_target is an array used to store pointers to WST objects that a PALLET object has been assigned to. The sequence of pointers in WST_target matches the sequence of WST objects a PALLET object must visit as it traverses the layers. A current Direction index references the link a pallet is traveling on, and a current WST_target index references the immediate workstation a pallet is trying to reach. As will be elaborated upon later, a PALLET object's indexes for the Direction and WST_target arrays are kept current through the use of its tick operation. Figure 3-6 shows the locations within a layer that a typical pallet is ticked in order to facilitate the chosen control architecture.

Attribute Name	Value Type	Description
dir_len	int	Number of LNK *'s in Direction list
dir_ind	int	Direction list index
Direction	**LNK	Direction list
wst_len	int	Number of WST *'s in workstation target list
wst_ind	int	Workstation target list index
WST_target	**WST	Workstation target list
d_need	int	Need direction flag: 1=need, 0=do not need
w_need	int	Need workstation flag: 1=need, 0=do not need
last_lnk	*LNK	Last link to tick pallet
order	int	Order number pallet's part belongs to
part	int	Part type of pallet's part
out	*OUT	Pointer to OUT pallet is assigned to
du	TCK	Due date of order pallet belongs to
pr	int	Priority of pallet

Table 3.3: Attributes of PALLET

Operation	Description
LNK*next_link()	Returns pointer to LNK assignment
int dir_need()	Returns d_need
WST * next_wst()	Returns pointer to next WST assignment
int wst_need()	Returns w_need
OUT * out_ptr()	Returns pointer to OUT pallet is assigned to
int order_num()	Returns order number pallet's part belongs to
int part_num()	Returns the part type number pallet is carrying
void tick(LNK*)	Update Direction and WST_target indexes
void fix_direc(int, LNK**)	Set Direction list
void fix_direc(int, WST**)	Set WST_target list
LNK * last_tick_on()	Returns last link to tick PALLET
TCK ck_due()	Returns pallet due_date
void priority(int)	Set pallet priority
int priority()	Returns pallet priority

Table 3.4: Operations of PALLET

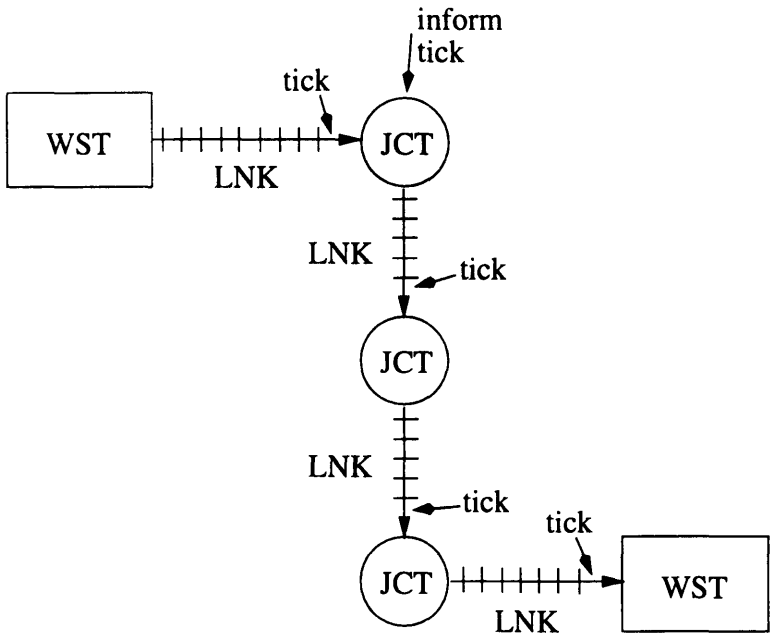

Figure 3-6: Junction and Link tick Locations

3.7 System-Level Component Structure

3.7.1 Input Stream Model

The object diagram for abstract data type IN is shown in Figure 3-7. IN has three attributes; pallet, Products, and L. An IN object's Products attribute is used to identify which products within the MPMS system's associated product family may be initiated from the IN object. This attribute is important when modeling systems in which the arrangement of links and workstations may not allow a pallet to reach every workstation it needs from every input stream. The Products attribute is implemented as an array with as many elements as there are products in the MPMS system's associated product family. The value of the ith element of the array is either 1 or 0; 1 if product i can use a particular input stream, and 0 otherwise. Class IN's member function part takes a product identification number as its argument and returns the ith element of the array implementing Products. IN's second attribute, pallet, implemented simply as a PALLET pointer, is used to model the space occupied by a pallet when it is loaded onto an input stream. Class IN's third attribute, L, is the pointer to the input layer-controller that oversees the loading of PALLET pointers onto the IN objects, as well as discharging the PALLET pointers downstream. The class IN member functions access and move are used to accomplish the task of loading the PALLET

pointers. Once a PALLET pointer is in an IN object, the class IN member function discharge is used by the input layer-controller to move it downstream.

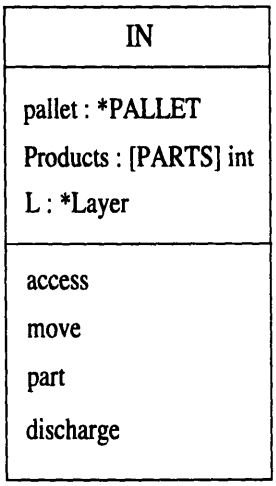

Figure 3-7: Object Diagram for Abstract Data Type IN

3.7.2 Link Model

The object diagram for abstract data type LNK is shown in Figure 3-8. Class LNK has four attributes; link, lnth, L, and index.

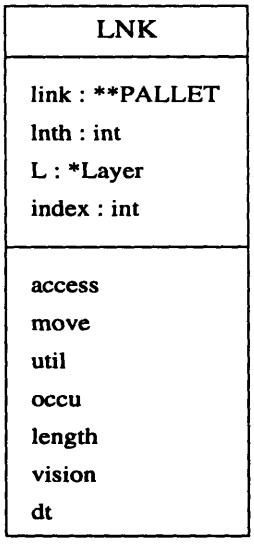

Figure 3-8: Object Diagram for Abstract Data Type LNK

The attributes link, and lnth are used to model the physical structure of a link. link is implemented as an array of PALLET pointers of length lnth. Hence, a pallet progressing

through a link at an unobstructed rate of one pallet length per second is implemented as a PALLET pointer being shifted one array element per second, in the direction of increasing array elements. In this implementation, a zero-valued pointer in the array means one pallet length of free space on the link. Class LNK member function dt() models the behavior of the link at each time increment. A flow chart of LNK::dt() is shown in Figure 3-9. Note that when a PALLET pointer reaches the end of a LNK object, the PALLET it points to is ticked. A LNK object's attribute L is the pointer to its associated layer-controller. Class LNK implements several operations which are available to its layer-controller. Class LNK member function util returns the percentage of a LNK object's link that is occupied with nonzero PALLET pointers. Member function occu returns the number of nonzero PALLET pointers in a LNK object's link. Member function length returns the lnth of a LNK object. Finally, member function vision takes an integer argument k, and returns the value of the kth element of a LNK object's link attribute, therefore allowing the layer-controller access to a PALLET pointer while it is still in the LNK object.

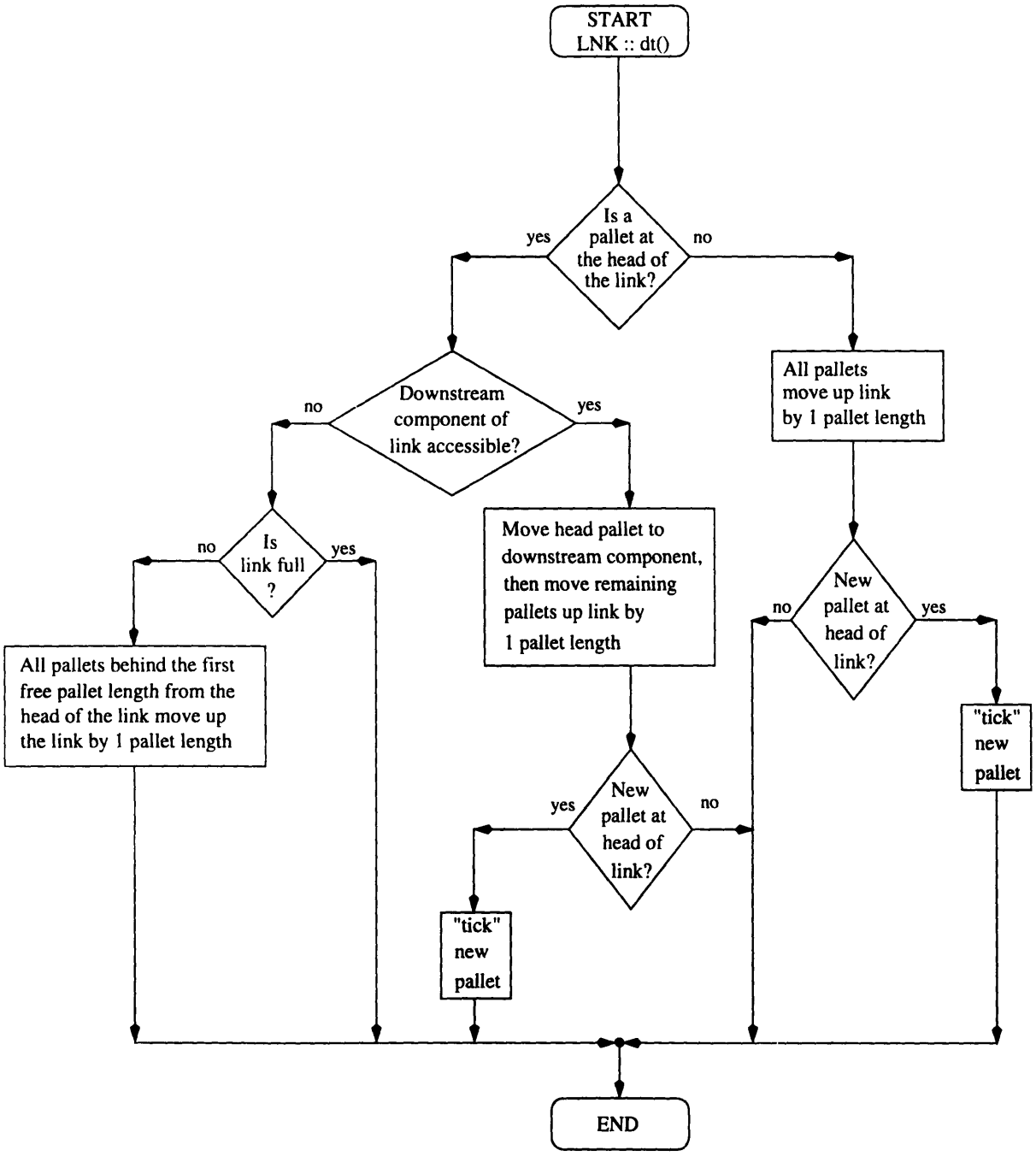

Figure 3-9: Flow Chart of LNK Behavior at each Time Step

3.7.3 Junction Model

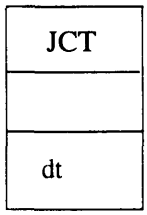

Figure 3-10: Object Diagram for Abstract Data Type JCT

The primary operation performed by the junction is to act as an intersection between multiple links. The object diagram for JCT is shown in Figure 3-10. As previously mentioned in the implementation of class JCT_base, the upstream linked list UP_JCT has two iterators. One of the iterators is specifically dedicated to implementing a method for establishing a precedence for pallets using a junction, should more than one pallet need to use the junction simultaneously. Figure 3-11 shows the flow diagram for the member function of class JCT, called dt. Class JCT member function, dt, automates the transfer of pallets from the junction's holding locations to the downstream links by implementing the precedence method. The iterator dedicated to the precedence relationships cycles through the links in the upstream linked list. Each time the expression "A := next upstream link with precedence" is used in the flow diagram, the iterator references the next link (JCT_UP_LINK) in the upstream linked list. The LNK object referenced in that link, through the value of its dat attribute is set equal to A.

Another operation performed by the junction allows a pallet entering a junction to communicate with the layer-controller to get directions, if necessary. This is implemented by the virtual function move. When a pallet is first moved onto a junction, move checks to see if the pallet needs directions. If a pallet needs directions, move contacts its layer-controller. The layer-controller then directly communicates a set of directions to the pallet. At minimum, the set of directions will have a list of links that the pallet must pass through to get to its assigned workstation or output stream. The layer-controller can also assign a set of workstations that the pallet must visit, if needed. Because the set of directions that the pallet received from the layer-controller start with the upstream link that the pallet just came from (the first link encountered in the current layer), the pallet must be ticked so its Direction index references the next link the pallet needs to use. See Figure 3-6.

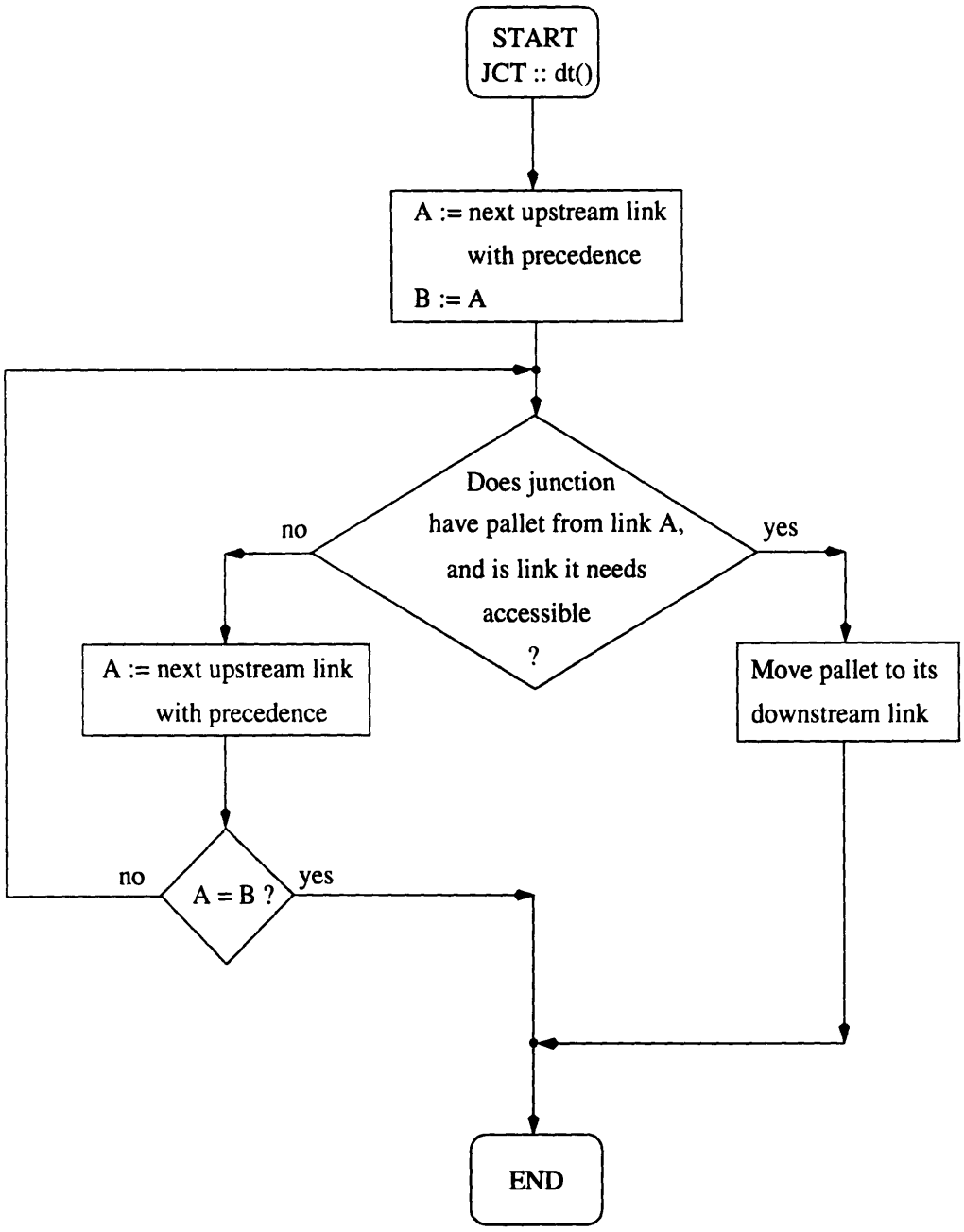

Figure 3-11: Flow Chart of JCT Behavior at each Time Step

3.7.4 Workstation Model

Pallets are processed at workstations. A workstations may or may not contain a buffer. Workstations are flexible, meaning each of their machines is capable of performing multiple tasks. Figure 3-12 shows the object diagram for WST. Class WST is implemented as an aggregate class with two component subclasses MACHINE and BUFFER, the C++ implementations of the machine and buffer, respectively.

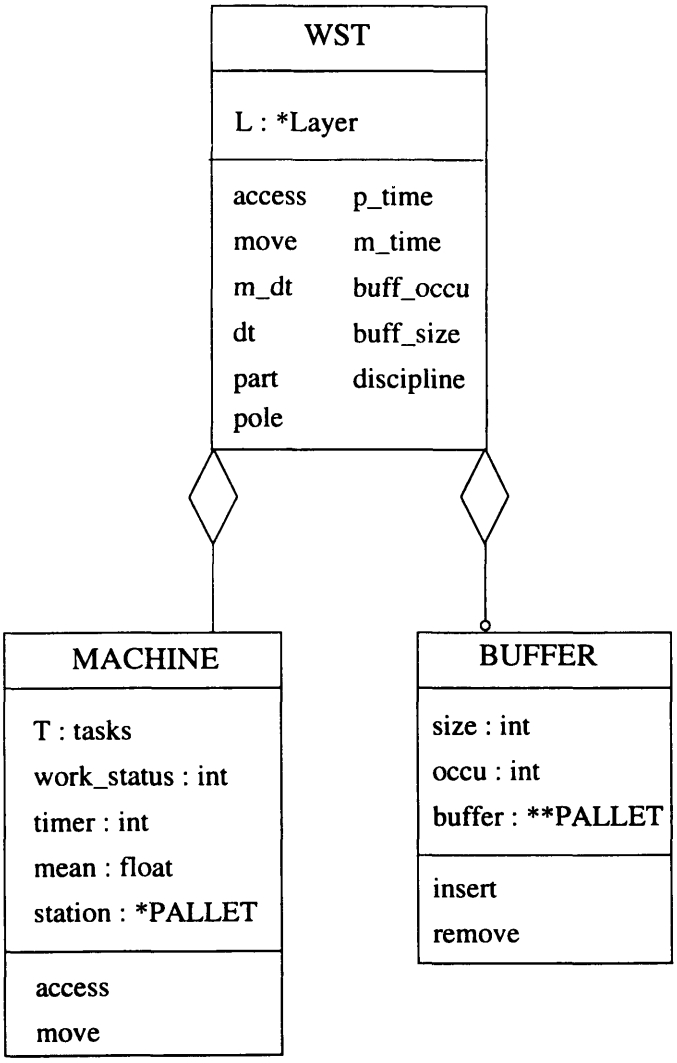

Figure 3-12: Object Diagram for WST Aggregation

Class MACHINE has five attributes; T, work_status, timer, mean, and station. T is implemented using the data structure tasks, which has the following definition:

```
struct tasks{int part[n]; int p_time[n];};
```

The first field in the structure tasks, part, is an array of integers of length n. n is the number of products in the MPMS system's associated product family. The ith element of the part array may be either 0, or 1; 0 if product i cannot use the workstation, and 1 if it can. The second field in the structure tasks, p_time, is an array of integers of length n. The ith element of array p_time is the amount of time in seconds that the workstation machine takes to perform its task on product i. It is important to note that elements of p_time might have the value 0. If in the implementation of a machine, the ith element of p_time has value 0 and the ith element of part has value 1, a pallet carrying product i may pass through the machine's workstation without undergoing processing. Here, the ith element of part having value 1 assures pallet i that the workstations necessary to complete its product will be accessible if it passes through this workstation. The MACHINE attribute work_status also takes on values of 0 or 1. A workstation uses work_status to determine if its machine is currently processing a pallet. work_status has value 1 when a part is being processed and 0 when the workstation machine is idle. The attribute timer contains the value equal to the number of seconds remaining for the machine to finish processing a pallet. At each time step, timer is decremented by one. This models the action of a workstation machine processing a pallet. When timer is 0, the task has been completed and work_status is set to 0. The next attribute of MACHINE is mean. Its value represents the average time in seconds it takes the machine to operate on products in the product family. Finally, the space occupied by a pallet being processed in the machine is modeled as the attribute station. station is implemented as a PALLET pointer. MACHINE has two functions, access and move, used to move a pallet onto the machine's processing station. Note that access and move are not virtual functions. The use of the names access and move are acceptable because class MACHINE does not inherit from abstract base class COMP.

BUFFER is implemented with three attributes; buffer, size, and occu, and two member functions; insert and remove. Attribute buffer models the locations where pallets are stored in the buffer. It is implemented as an array of PALLET pointers. The value of the attribute size is the pallet capacity of the buffer. Attribute occu is the number of nonzero PALLET pointers in buffer. BUFFER's two operations, insert and remove, model respectively, the insertion and removal of pallets from the buffer.

Class WST's attribute L is the pointer to the workstation's associated central layer-controller. WST has eleven functions. The implementation of virtual functions, access and

move, was described previously. The function pole is defined by the simulation programmer. pole is used by layer-controllers to poll their workstations as to how long they require to process a given pallet. The value returned by pole reflects how long it would take a workstation to service the pallet if it were moved to the workstation at the time the poll was taken. Functions part, p_time, and m_time are used by the layer-controller to access information on the workstation's product processing ability. The function part has an integer argument k. part returns 1 if part k can be handled by the workstation machine, and returns 0 otherwise. p_time returns the amount of time in seconds it takes the workstation machine to process the pallet. m_time returns the value of the machine's attribute mean. The three functions: buff_sz, buff_occu, and discipline relate to the buffer. buff_sz returns the number of pallets the workstation buffer can hold and buff_occu returns the number of pallets currently in the buffer. The function, discipline is specified by the simulation programmer and is used to decide which pallets are to be moved next from the buffer to the machine, and under what conditions. The remaining two functions in WST, m_dt and dt implement the behavior of the workstation at each time step. m_dt is used to implement the behavior of the machine at every time interval and is used by dt. Figures 3-13 and 3-14 show the flow diagrams from which dt and m_dt were implemented.

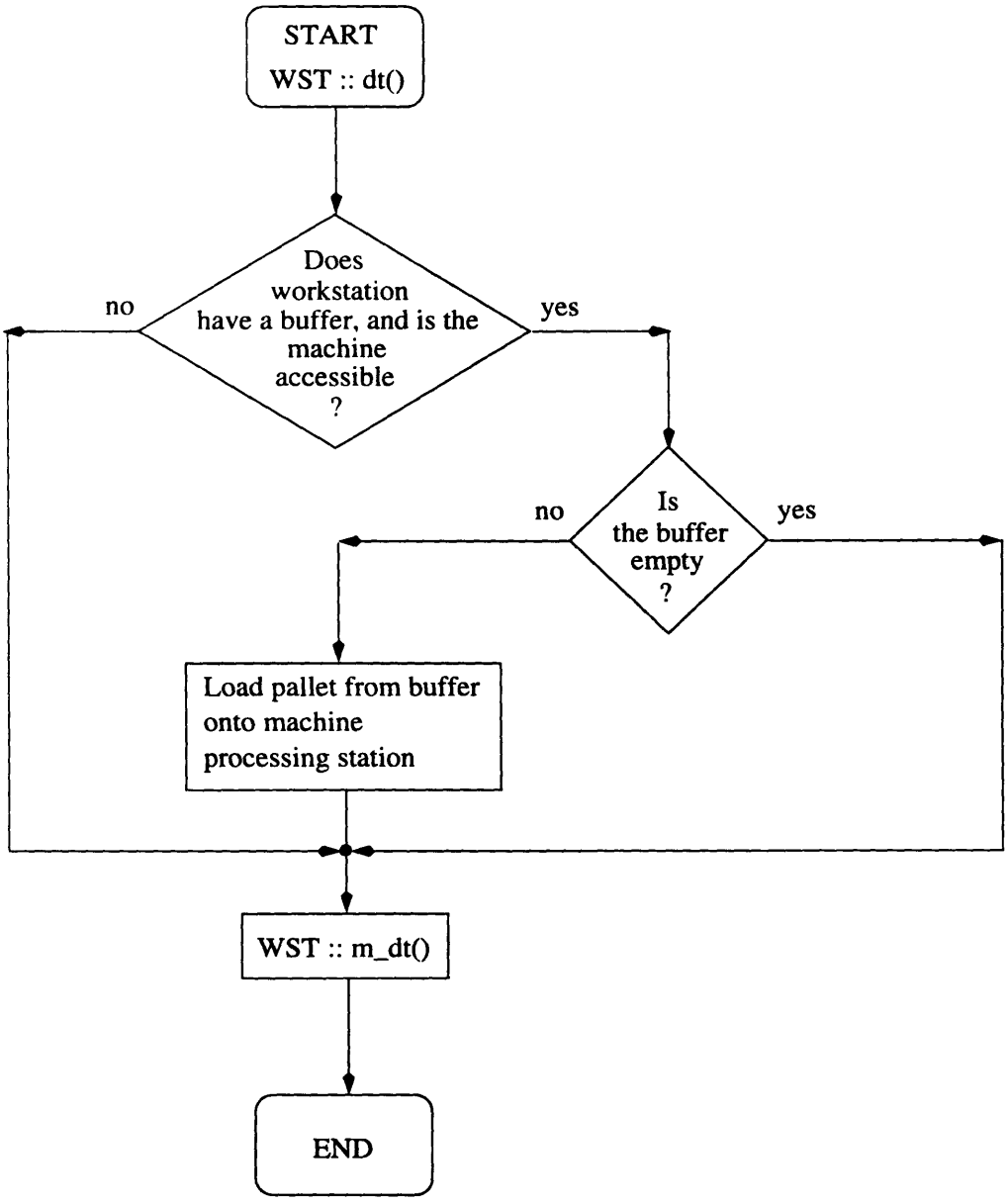

Figure 3-13: Flow Chart of WST Behavior at each Time Step

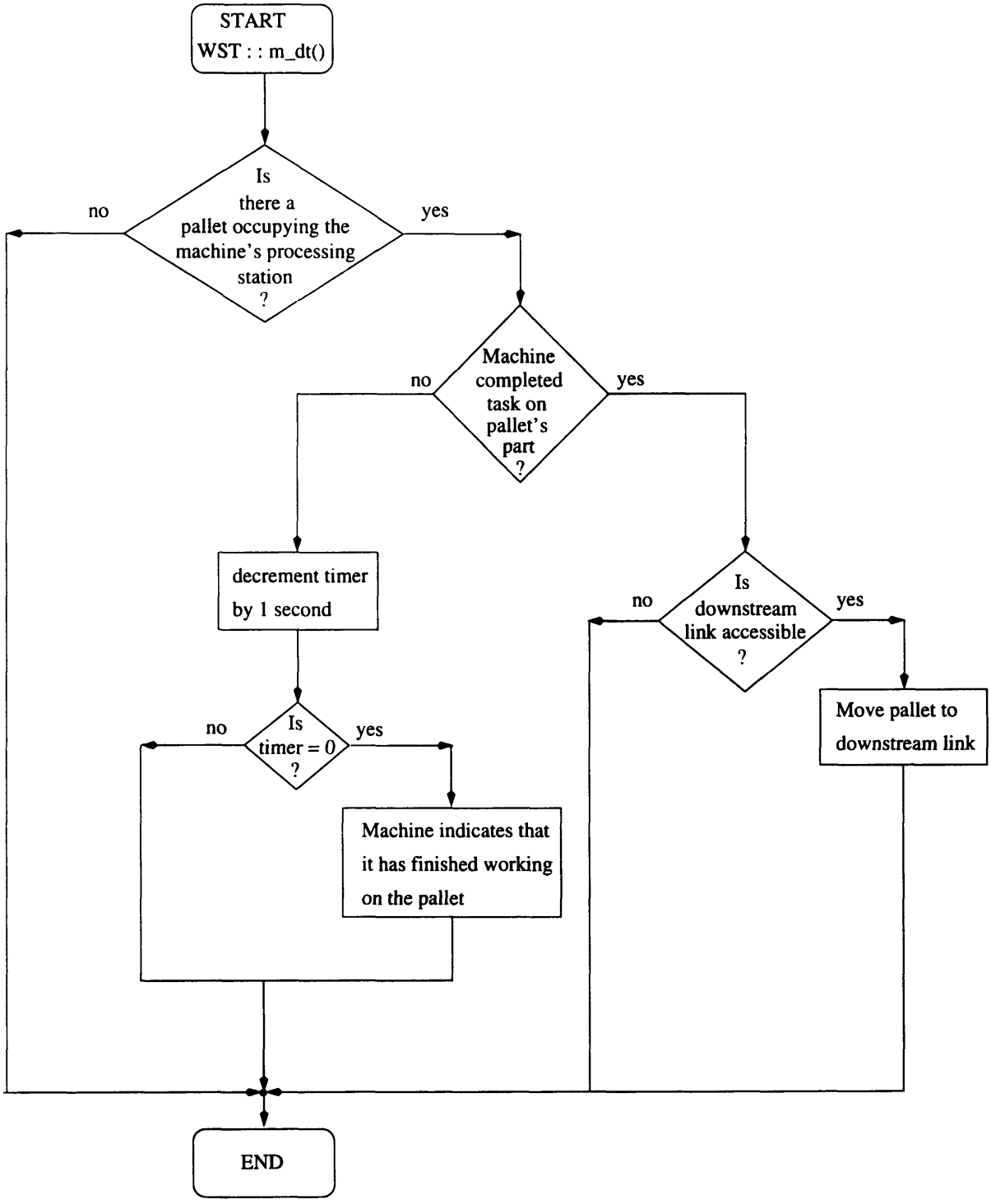

Figure 3-14: Flow Chart of MACHINE Behavior at each Time Step

Attribute Name	Value Type	Description
order_id	int	Id number of the customer order at OUT_ORDER
due_date	TCK	Due date of customer order at OUT_ORDER
order_sz	int	Number of products in customer order
completed_sz	int	Number of products in completed customer order at OUT_ORDER
order_comp	COMPOSITION	Composition of customer order
completed_comp	COMPOSITION	Composition of completed customer order at OUT_ORDER

Table 3.5: Attributes of Class OUT_ORDER

3.7.5 Output Stream Model

The final system-level component to be discussed is the output stream. Figure 3-15 shows the object diagram for the abstract data type OUT which models the output stream. Class OUT is implemented as an aggregate class containing subclass OUT_QUEUE. An OUT_QUEUE object is modeled as a singly-linked list of OUT_ORDER objects. In a simulation, information about each customer order assigned to an OUT object is recorded in one of its OUT_ORDER objects. The attributes and operations of class OUT_ORDER are recorded respectively in Tables 3.5 and 3.6. The functions of class OUT_QUEUE are only accessible to class OUT. They are used in the implementation of the member functions of class OUT. Class OUT has three attributes; o_num, p_num and L. In the implementation of an output stream, the attribute o_num indicates the number of customer orders in the system input queue which are assigned to the input stream and have not yet been completed. The value of p_num, is equal to the number of pallets from the customer orders assigned to the output stream which have not yet reached the output stream. Finally, the attribute L is a pointer to the output layer-controller. The pointer is needed so that an output stream can inform the output layer-controller that one of its customer orders has been completed.

Operation	Description
int id()	Returns value of order_id
TCK ck_due()	Returns value of due_date
int o_size()	Returns value of order_sz
int c_size()	Returns value of completed_sz
int o_comp(int N)	Returns value of Nth element of order_comp
int c_comp(int N)	Returns value of Nth element of completed_comp
int change_comp(int N, int M)	Changes the value of Nth element of completed_comp by M, and changes the value of completed_sz by M

Table 3.6: Operations of Class OUT_ORDER

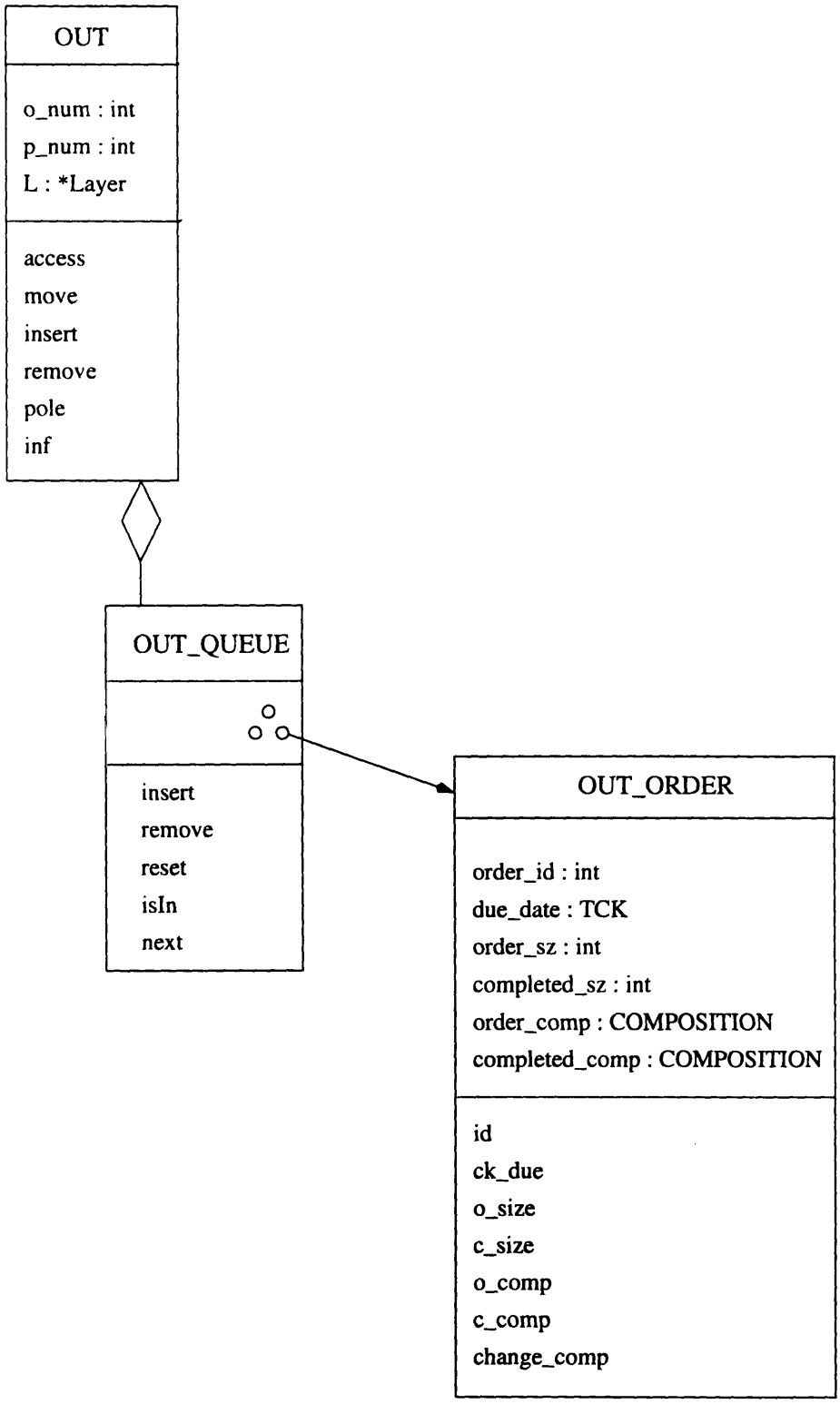

Figure 3-15: Object Diagram for OUT, OUT_QUEUE, and OUT_ORDER

In an OUT object, the function insert creates an OUT_ORDER object when it is assigned a customer order. Member function pole is used by the layer-controller responsible for assigning output streams to newly registered customer orders. This function is defined by the simulation programmer to reflect information deemed relevant in the assignment of output streams. Member function inf takes as its argument the identification number of a customer order belonging to an OUT object and returns the pointer to the OUT_ORDER object for the customer order. Virtual function move has behavior which is specific to the output stream. When a pallet is moved into an output stream, move modifies the information in the OUT_ORDER object of the customer order the pallet belongs to. Should the order be completed when a pallet is moved into an output stream, move informs the output layer-controller of this event. For programming purposes, move also deletes the PALLET object.

3.8 Modeling the MPMS System Layer-Controllers

As modeled, the MPMS system-level components interact with one another through the use of the virtual functions previously described. Moreover, any system-level component may only communicate directly with its neighboring components. Because of the communications limitation between system-level components, layer-controllers have been defined to coordinate the flow of pallets and information within each system layer. To help accomplish this, each layer-controller can directly access the operations and public attributes of all the system-level components within its layer. Should routing decisions require information from system-level components distributed throughout multiple system layers, then routing decisions are coordinated through the use of a control network which can access the necessary layer-controllers. Each layer-controller also has information as to how its system-level components are interconnected. This allows the layer-controller, for example, to know which links a pallet must pass through to reach a workstation from a junction.

Three types of layer-controllers are defined for the MPMS system: The input, central, and output layer-controllers. Figure 3-16 shows the object diagram for the layer-controller abstract generalization hierarchy. Abstract base class Layer contains no attributes and two operations. The descendants of class Layer; Input_Layer, Central_Layer and Output_Layer are the C++ implementations of the input, central, and output layer-controllers, respectively.

The implementation of each system-level component contains a Layer pointer so that it may request operations from its respective layer-controller. Because of this, the operations of class Layer must be implemented as virtual functions. The operation inform is used by a junction to notify its layer-controller that it contains a pallet which needs directions. Since junctions are only found in central and output layers, virtual function inform must only be overridden in classes Central_Layer and Output_Layer. The operation done is used by an output stream to notify an output layer that a customer order has been completed. Hence, virtual function done is only overridden in class Output_Layer. Class Input_Layer has no implementation for either of the Layer virtual functions. In fact, as implemented, class Input_Layer inherits nothing from abstract base class Layer. Class Input_Layer was included in the hierarchy however, because it is a layer-controller and future embellishments to the hierarchy may allow for Layer operations that will be inherited by all three of its descendants.

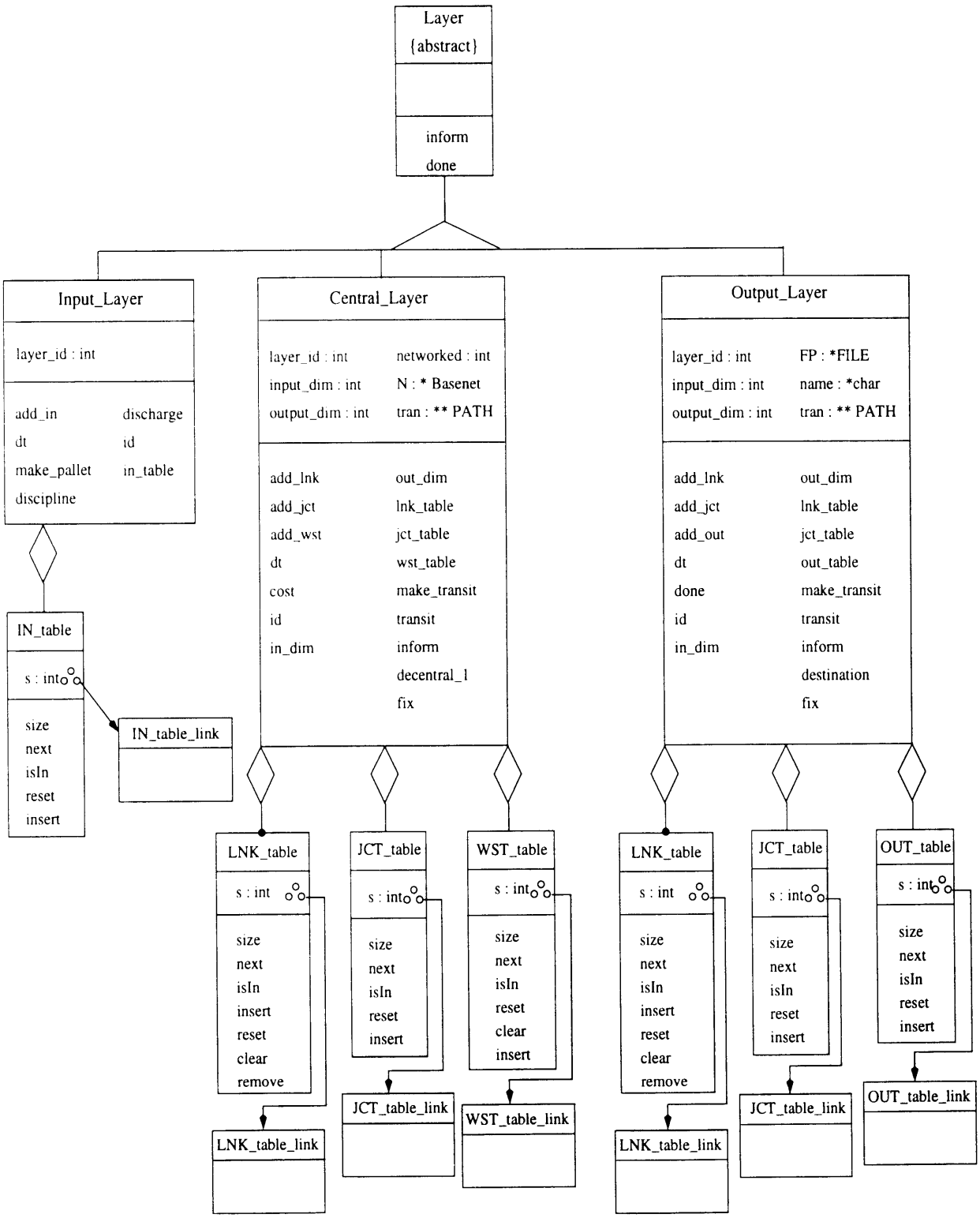

Figure 3-16: Abstract Data Type Layer Hierarchy

Operation	Description
size	Return number of Links in List
next	Return pointer to system-level component stored in current Link, move List iterator to next Link
isIn	1. Accepts argument of type typ, returns * to system-level component in List whose comp_typ attribute matches the argument, or zero if no match 2. Accepts a comp * to a component and down converts the pointer's type to a system-level component * e.g. * IN, * WST
insert	A new Link is inserted into the List, to store the argument
reset	Reset List iterator to first Link in List
clear	Delete all Links in List
remove	Remove first Link in List

Table 3.7: Layer Descendant Aggregate Operations

Each layer-controller can directly access the operations and public attributes of all the system-level components within its layer. This is achieved by defining the specialized layer-controller classes as aggregates. Each class contains subclasses that are implemented as singly-linked lists of pointers to its system-level components. As shown in Figure 3-17, each subclass is dedicated to one type of system-level component: Class IN_table is implemented as a singly-linked list of IN_table_link objects, each of which holds a pointer to an IN object pointer. Class JCT_table is implemented as singly-linked list of JCT_table_link objects, each of which holds a pointer to a JCT object pointer. Class LNK_table is implemented as a singly-linked list of LNK_table_link objects, each of which holds a pointer to a LNK object pointer. Class WST_table is implemented as a singly-linked list of WST_table_link objects, each of which holds a pointer to a WST object pointer. Class OUT_table is implemented as a singly-linked list of OUT_table_link objects, each of which holds a pointer to an OUT object pointer. For each class implemented as a singly-linked list, the value of s is equal to the number of links an instance of that class may have. Also, for each of these classes, operations with the same name behave essentially the same. Table 3.7 defines these operations. In this table, List refers to any object implemented as a singly-linked list, and Link, any object acting as a link in the list.

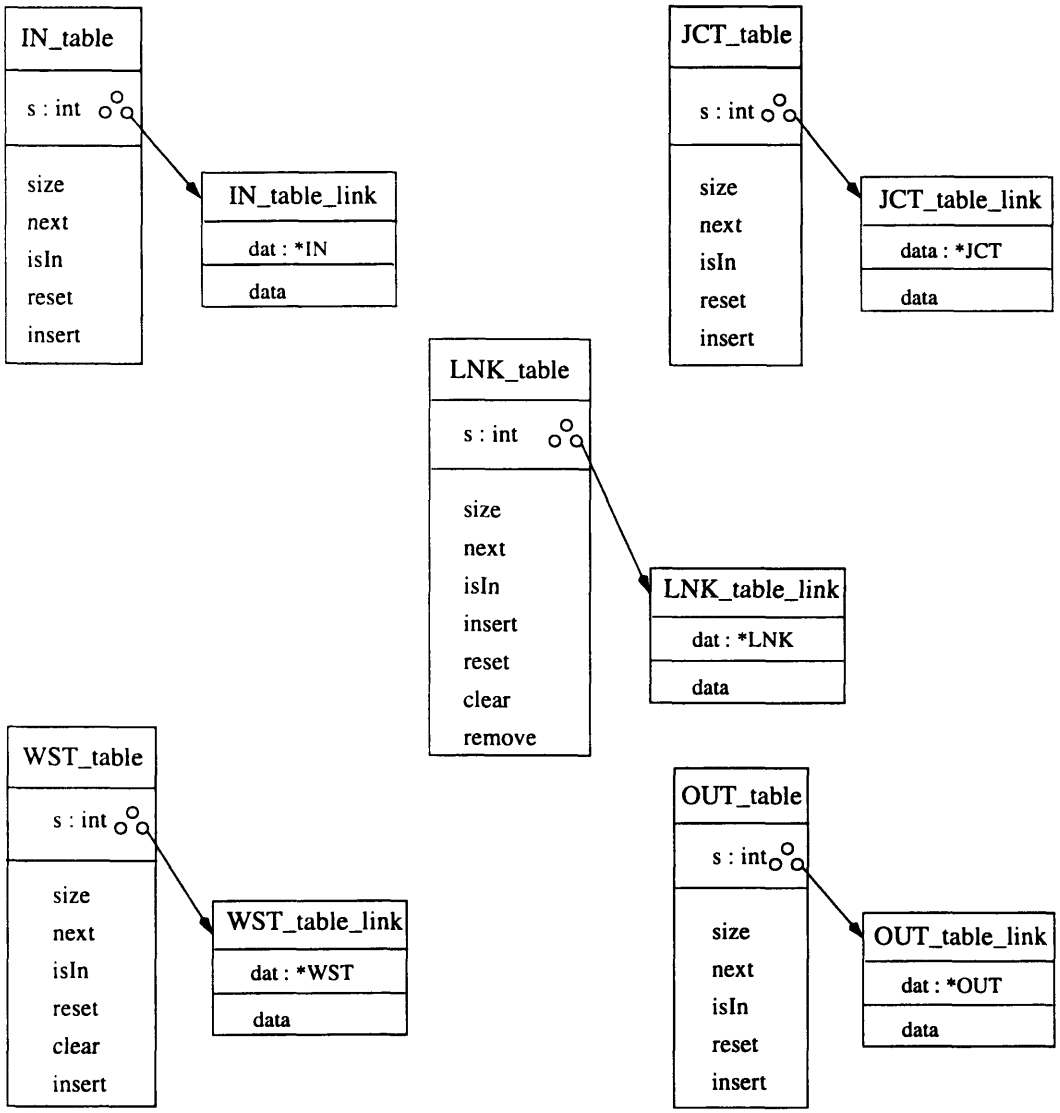

Figure 3-17: Object Diagrams of Layer Descendant Aggregates

3.8.1 Modeling the Input Layer-Controller

Class Input_Layer has one attribute. The value of layer_id, is the input layer's identification number. The seven operations of class Input_Layer are given in Table 3.8. The data structure Next is the return value of discipline, and the argument of make_pallet. Next is defined as:

```
struct Next{int part; IN_ORDER * in_order;};
```

When returned by discipline, Next's first field specifies the product number which should be loaded onto the IN object whose pointer was discipline's argument. Next's second field

Operation	Description
IN * add_in(IN*n)	Insert n into IN_table, return n
void dt()	Time step behavior of Input_Layer
PALLET * make_pallet(next m)	Creates a pallet specified by m, returns pallet's *
int id()	Returns layer_id
IN_table * in_table()	Returns * to Input_Layer object's IN_table object
next discipline(IN*, IN_QUEUE&)	Specified by simulation programmer
int discharge(IN*)	Specified by simulation programmer

Table 3.8: Operations of Class Input_Layer

is the pointer to the in_order object that the product came from. The function discharge takes an IN object's pointer as its explicit argument and returns either a 0 or a 1. A 1 signals that the pallet at the IN object is allowed to move downstream, a 0 means the pallet must remain at the IN object. This function is useful when the conditions necessary for discharging a pallet from its input stream depends on more than whether or not the input stream's downstream link is accessible. The functions discipline and discharge are specified by the simulation programmer. The final Input_Layer function, dt, implements the behavior of the input layer-controller at each time step. The flow chart for dt is shown in Figure 3-18.

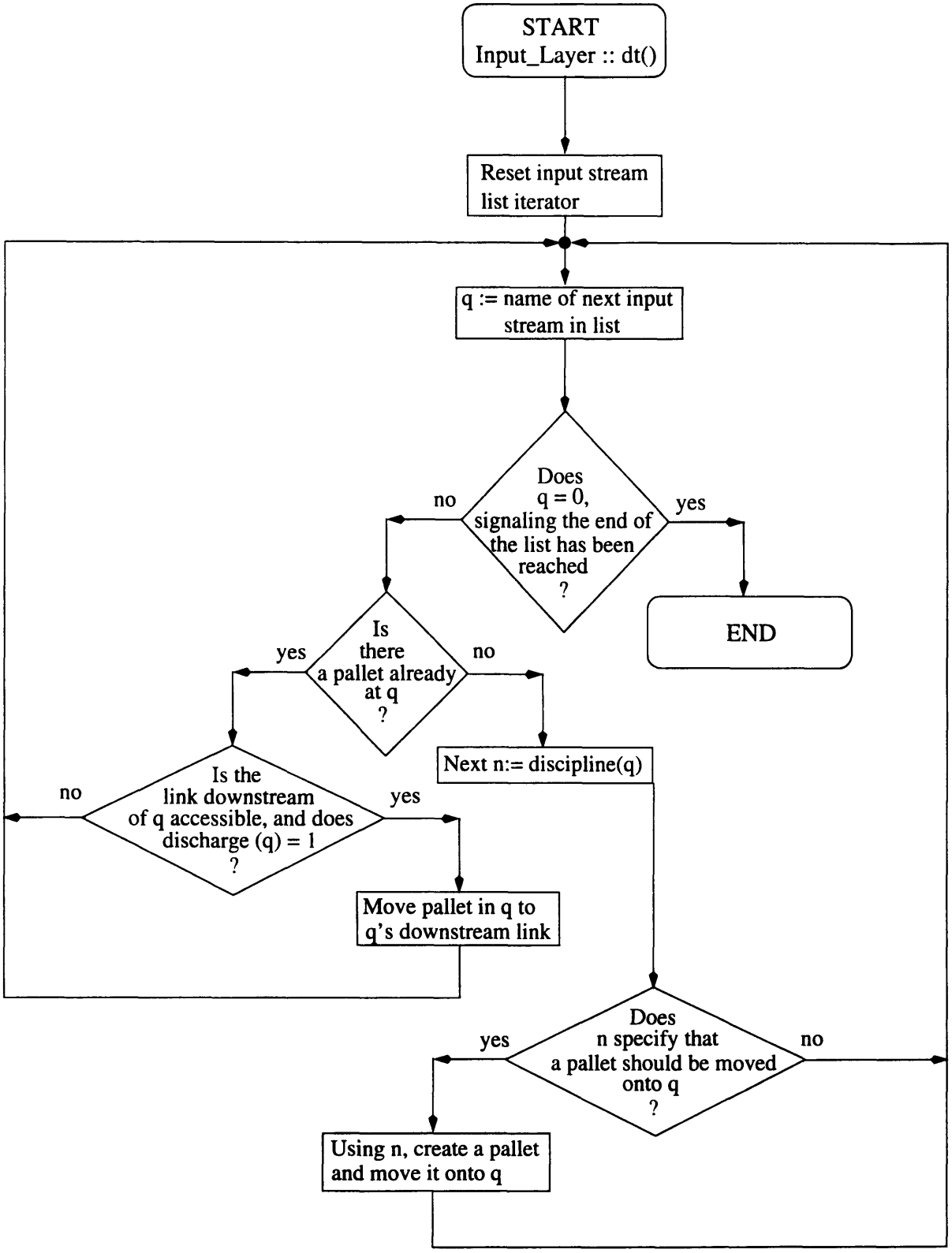

Figure 3-18: Flow Chart of Input_Layer Behavior at each Time Step

Attribute Name	Value Type	Description
layer_id	int	Layer identification number
input_dim	int	Number of input links in layer
output_dim	int	Number of workstations in layer
networked	int	1 ⇒ pallet routing decisions handled by Network
		0 ⇒ pallet routing decisions handled locally
N	*Basenet	Pointer to Network
tran	**PATH	2×2 array of PATH objects

Table 3.9: Attributes of Class Central_Layer

3.8.2 Modeling the Central Layer-Controller

The attributes of class Central_Layer and their meanings are shown in Table 3.9. The Central_Layer attribute tran is a 2-dimensional array of PATH objects. The number of rows in the array is equal to the value of output_dim, and the number of columns is equal to the value of input_dim. Class PATH has been defined to facilitate the assignment of LNK object pointer sequences. The class has two public attributes, list and size. list is implemented as an array of LNK object pointers, and the value of size is equal to the number of pointers in the array. For a given Central_Layer object, each PATH in tran is associated with a link-workstation pair in the central layer, where the link's upstream component is in another layer. Such a link will be referred to as an input link. After all system-level component objects in a layer have been created and joined, the function make_transit() can generate tran, provided that the following condition is satisfied:

> For any central layer, there exists at most one sequence of links a pallet can pass through to get from an input link to a workstation, while never coming in contact with any junction more than once.

If such a sequence of links exists, then the PATH object in tran associated with the input link–workstation pair records the sequence in its list attribute. The value of its size attribute is equal to the number of elements in the sequence. Should no sequence of links exist from an input link to a workstation, the associated PATH object in tran will have a value of 0 for both its attributes. The function transit(LNK * L, WST * W) returns a pointer to the PATH in tran associated with the object pointers W and L, where L points to an input LNK object.

The flow diagram of the behavior of virtual function inform is shown in Figure 3-19. This function is used by a junction to inform its central layer-controller that it has a pallet which needs directions. A junction does not know what kind of directions a pallet needs, this is determined by the central layer-controller. As shown in Figure 3-19, the layer- controller first determines whether a pallet needs to be immediately assigned to a workstation. If the pallet already has a next workstation assignment, then the layer-controller gives the pallet the path it must follow to reach the workstation. Should the pallet have no assigned workstations, there are two scenarios which must be considered. In the first scenario, the decision for assigning a workstation to the pallet involves only information within the scope of the central layer-controller. In this scenario, a pallet is only assigned one workstation, namely the one in the current layer that it will visit next. The flow diagram of the function decentral_1 used to make this workstation assignment is shown in Figure 3-20. The function cost used in decentral_1 must be defined by the simulation programmer.

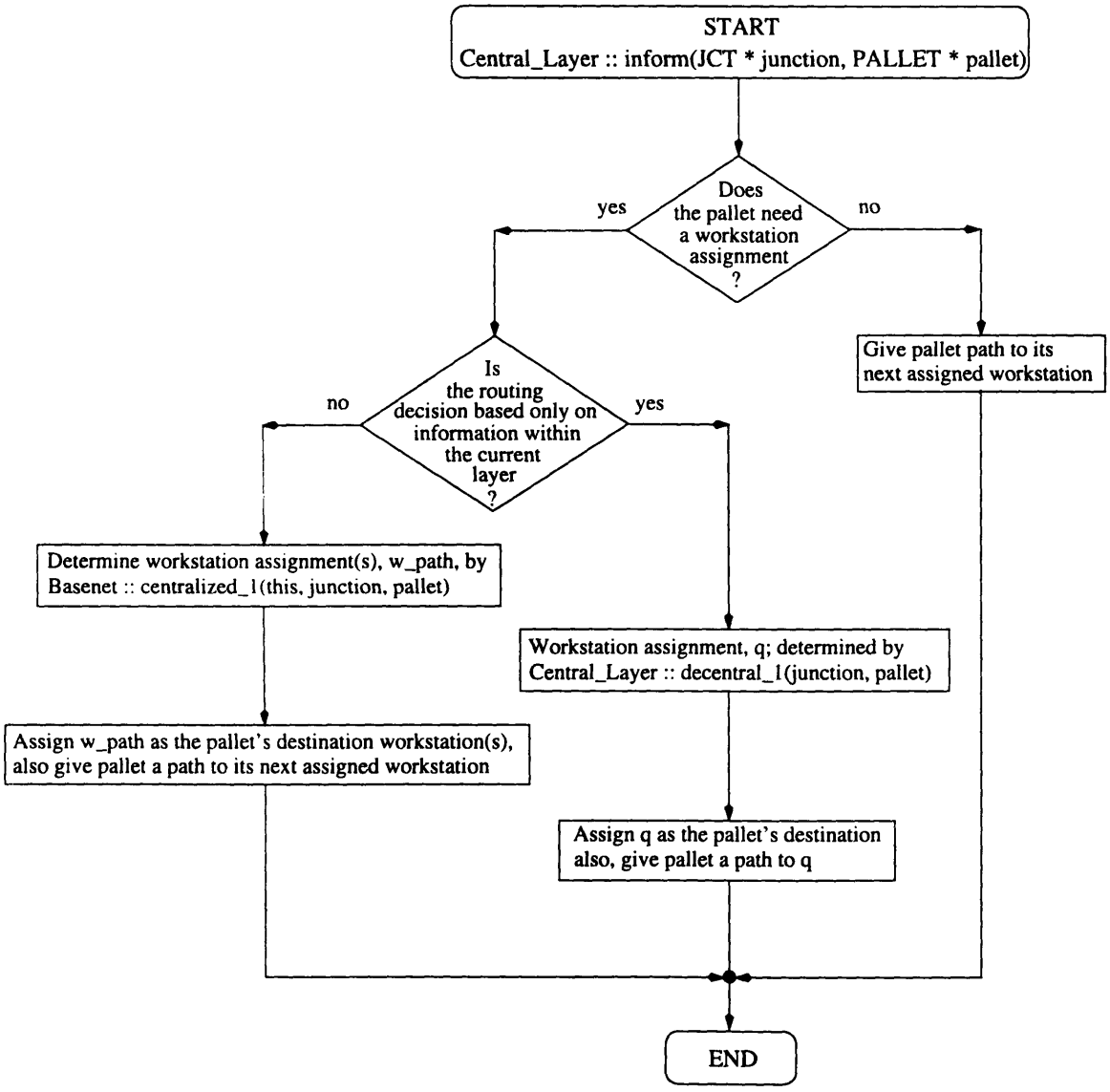

Figure 3-19: Flow Chart of Central_Layer::inform

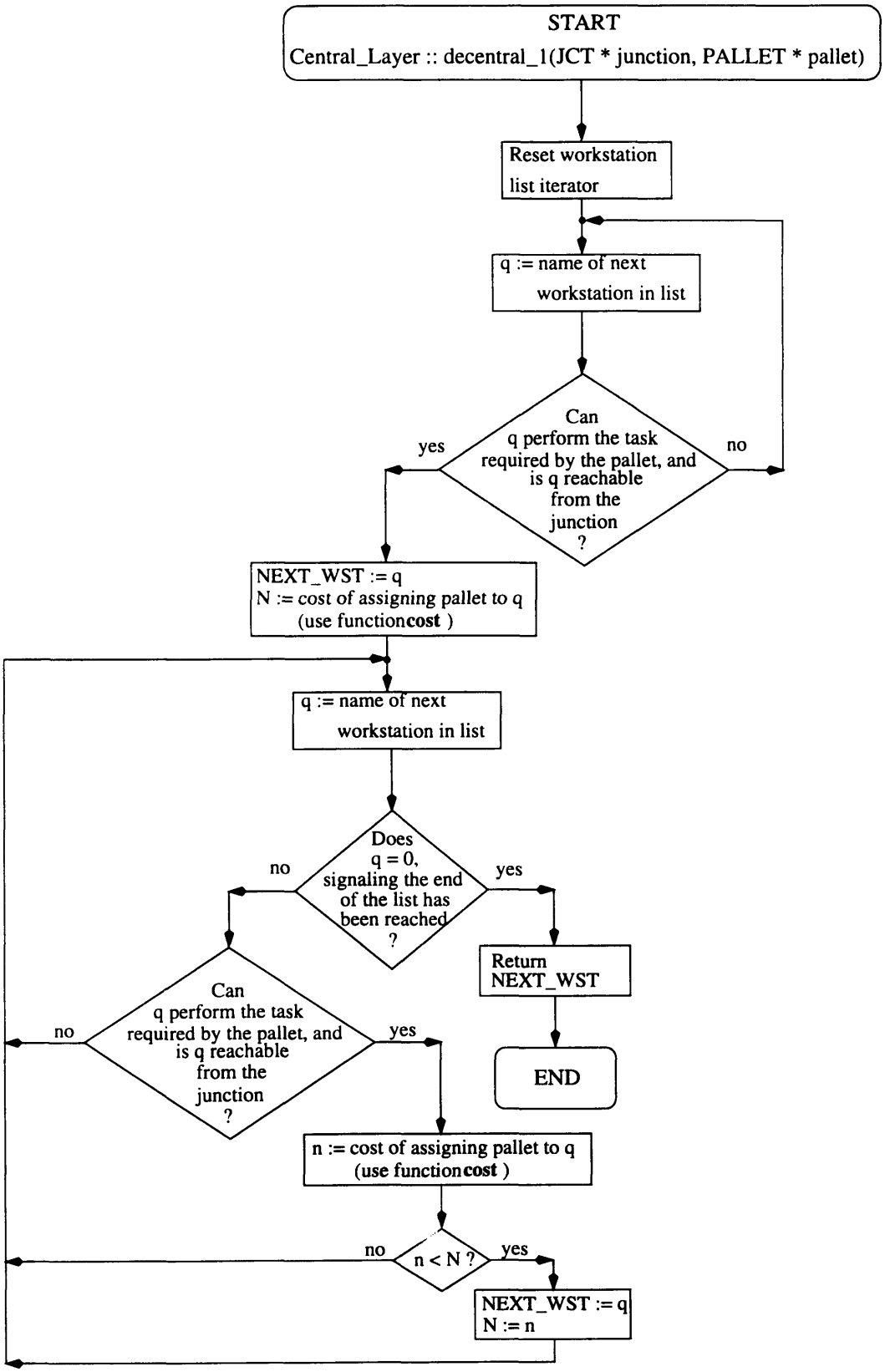

Figure 3-20: Flow Chart of Central_Layer ::decentral_1

In the second scenario, the decision for giving a pallet its workstation assignment for the current layer, and possibly several more workstations after that, involves information not restricted to the scope of the layer-controller. The additional workstations assigned are the workstation a pallet will visit in successive layers, one workstation per layer, starting from the next layer.

The class W_PATH has been defined to make the assignment WST object pointers easier. Its definition is almost exactly the same as class PATH, with the exception that PATH holds LNK object pointers, and W_PATH holds WST object pointers. The remaining operations of class Central_Layer are given in Table 3.10. The class W_PATH has been defined to make the assignment WST object pointers easier. Its definition is almost exactly the same as class PATH, with the exception that PATH holds LNK object pointers, and W_PATH holds WST object pointers. The remaining operations of class Central_Layer are given in Table 3.10.

3.8.3 Modeling the Output Layer-Controller

The attributes of class Output_Layer and their meanings are shown in Table 3.11. The Output_Layer attribute tran is a 2-dimensional array of PATH objects. The number of rows in the array is equal to the value of output_dim, and the number of columns is equal to the value of input_dim. Each PATH in tran is associated with an input link–output stream pair in the output layer. After all system-level component objects in the output layer have been created and joined, the function make_transit() can generate tran, provided that the following condition is satisfied:

> In the output layer, there exists at most one sequence of links a pallet can pass through to get from an input link to an output stream, while never coming in contact with any junction more than once.

If such a sequence of links exists, then the PATH object in tran associated with the input link and the output stream records the sequence in its list attribute. The value of its size attribute is equal to the number of elements in the sequence. Should no sequence of links exist from an input link to an output stream, the associated PATH object in tran will have a value of 0 for both its attributes. The function transit(LNK * L, OUT * O) returns a pointer to the PATH in tran associated with object pointers L and O, where L points to an input LNK object.

Operation	Description
LNK * add_lnk(LNK * L)	Add LNK pointer to LNK_table
JCT * add_jct(JCT * J)	Add JCT pointer to JCT_table
WST * add_wst(WST * W)	Add WST pointer to WST_table
void dt()	Execute dt() of every object addressed in LNK_table, JCT_table, and WST_table
void make_transit()	Generate tran
float cost(PATH * D, PALLET * P, WST * W)	Simulation programmer defined function
int id()	Returns value of layer_id
int in_dim()	Returns value of input_dim
int out_dim()	Returns value of output_dim
LNK_table * lnk_table()	Returns pointer to LNK_table
JCT_table * jct_table()	Returns pointer to JCT_table
WST_table * wst_table()	Returns pointer to WST_table
PATH * transit(LNK * L, WST * W)	Returns pointer to PATH in tran associated with LNK and WST pointers
void inform(JCT * J, PALLET * P)	Used by JCT to inform Central_Layer of a pallet which needs directions
WST * decentral_1(JCT * J, PALLET * P)	Implements algorithm used to assign a pallet its next WST
void fix(PALLET * P, PATH * D, WST * W)	Assigns pallet W_PATH and PATH to next WST
void fix(PALLET * P, PATH * D)	Gives pallet a PATH to its next WST
void fix(PALLET * P, PATH * D, W_PATH)	Assigns pallet next WST and PATH

Table 3.10: Operations of Class Central_Layer

Attribute Name	Value Type	Description
layer_id	int	Layer identification number
input_dim	int	Number of input links in layer
output_dim	int	Number of output streams in layer
FP	*FILE	File pointer
name	*char	Name of file dat is written to
tran	**tran	2×2 array of PATH objects

Table 3.11: Attributes of Class Output_Layer

Operation	Description
LNK * add_lnk(LNK * L)	Add LNK pointer to LNK_table
JCT * add_jct(JCT * J)	Add JCT pointer to JCT_table
WST * add_wst(OUT * O)	Add OUT pointer to OUT_table
void dt()	Execute dt() of every object addressed in LNK_table, JCT_table, and OUT_table
void make_transit()	Generate tran
void done(OUT * O, int i)	Used by OUT to inform Output_Layer that order i is complete
int id()	Returns value of layer_id
int in_dim()	Returns value of input_dim
int out_dim()	Returns value of output_dim
LNK_table * lnk_table()	Returns pointer to LNK_table
JCT_table * jct_table()	Returns pointer to JCT_table
OUT_table * out_table()	Returns pointer to OUT_table
PATH * transit(LNK * L, OUT * O)	Returns pointer to PATH in tran associated with LNK and OUT pointers
void inform(JCT * J, PALLET * P)	Used by JCT to inform Central_Layer of a pallet which needs directions
OUT * destination(IN_ORDER * CO)	Used to assign a customer order an OUT
void fix(PALLET * P, PATH * D)	Gives pallet a PATH to its OUT

Table 3.12: Operations of Class Output_Layer

The remaining operations of class Output_Layer are given in Table 3.12. The Output_Layer function destination is defined by the simulation programmer. It is used to determine which output stream a customer order will be assigned to when the order is registered in the customer order queue. The virtual function inform is used by a junction to inform the output layer-controller that it has a pallet which needs directions. The function causes the output layer-controller to give the pallet a path to its assigned output stream. When all the pallets associated with a customer order arrive at an output stream, the output stream uses the virtual function done to inform the output layer-controller. The output layer-controller then deletes the customer order's OUT_ORDER object from the output stream's OUT_QUEUE, and the InOrder object from the customer order queue, InQueue. For measurement purposes, information about the customer order will also be written to file name, as explained further in Chapter 4.

Operation	Description
int size()	Returns value of s
Central_Layer * insert(Central_Laver * CL)	Add Central_Layer * to net_table
Central_Layer * next()	Returns next Central_Layer * in net_table
void reset()	net_table iterator reset to first net_table_link
Central_Layer * isIn(int i)	Returns Central_Layer pointer to Central_Layer with layer_id i

Table 3.13: Operations of Class net_table

3.8.4 Modeling the Control Network

The object diagram for class Network is shown in Figure 3-21. In any simulation, only one instance each of classes Basenet, Network, and net_table will be implemented, so the instances will be referred to as Basenet, Network, and net_table respectively. Network is an aggregate class, whose subclass net_table is implemented as a singly-linked list of net_table_link objects. Each net_table_link object stores a pointer to a Central_Layer object. The net_table has one attribute, s, whose value is equal to the number of net_table_link objects it contains. The five functions of net_table are given in Table 3.13. Network has two attributes and two functions. The attribute, networked, was included for easy switching between scheduling policies A and B, as given in Chapter 4. The remaining Network attribute is used in the implementation of the function centralized_1. The two functions of Network are add_C and centralized_1. The first, add_C, is used to create a net_table_link for a Central_Layer object pointer, and insert it into the net_table. The second function, centralized_1, is defined by the simulation programmer. Its function is to assign workstations to a pallet as explained in the second pallet assignment scenario of the Central_Layer::inform function in section 3.8.2. centralized_1 is implemented as a virtual function which overrides Basenet's function. This was done because all Central_Layer objects are given a pointer to Basenet.

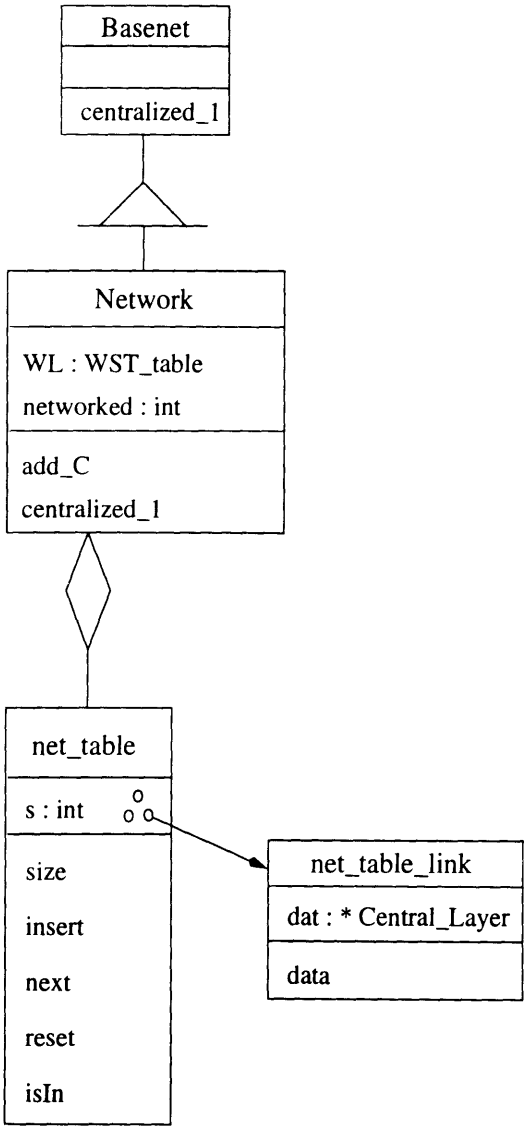

Figure 3-21: Object Diagram for Network

Chapter 4

Simple MPMS System Simulation

4.1 Introduction

In this chapter, a product family with 9 members will be defined, along with the layout for a simple MPMS system capable of producing members of the product family. Assumptions on the distribution and arrival rates of customer orders will be made so that the performance of the MPMS system under different scheduling policies may be compared. Specifically, two scheduling policies will be introduced to control the simple MPMS system. These policies will then be compared through simulation.

4.2 Product Family

Each member of the product family enters the MPMS system as a common part carried by a pallet. Workstations in the system perform tasks on a common part, and the resultant product depends on which tasks were performed. Exactly two tasks will be performed on each common part. The first task performed may be one of three different tasks T_0, T_1, and T_2. The second task performed may be one of the tasks T_3, T_4, T_5. Hence, the product family will have a total of 9 members. Table 4.1 shows the task sequence which is required to produce each member of the product family.

Product	Task Sequence
0	T_0, T_3
1	T_1, T_3
2	T_2, T_3
3	T_0, T_4
4	T_1, T_4
5	T_2, T_4
6	T_0, T_5
7	T_1, T_5
8	T_2, T_5

Table 4.1: Task Sequence Required to Produce Each Member of Product Family

4.3 MPMS System Layout

The simple MPMS system which will be simulated in this chapter is shown in Figure 4-1. The system is a member of the class of MPMS systems described in Chapter 2. It is composed of the five system-level components; input streams, links, junctions, workstations, and output streams. The guideway system, composed of links and junctions, follows the same pattern as the guideway system in Figure 2–1. Each pallet has no choice but to exit a central layer through a workstation. The system has three input streams in its input layer, three workstations in each of its two central layers, and three output streams in its output layer. All of these quantities were arbitrarily selected, with one restriction on the number of central layers.

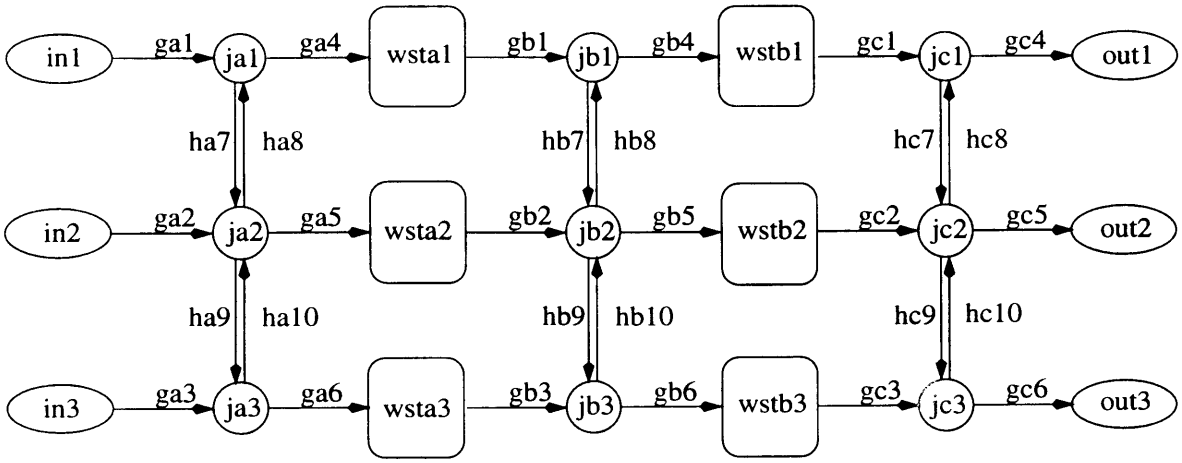

Figure 4-1: Layout of Simple MPMS System

This restriction is due to the fact that the system must be able to produce members of its

associated product family, each of which require two tasks. At most one task per central layer may be performed on a given pallet, therefore a minimum of two central layers is required. Note that in each central layer, paths exist to get from every input link to every workstation. Now that the basic layout of the MPMS system to be simulated has been specified, all that remains is to provide the specific parameters of the system-level components for this simulation. Tables 4.2– 4.7 provide these system-level component characteristics.

Input Stream	Products Handled
in1	all
in2	all
in3	all

Table 4.2: Input Stream Product Processing Capabilities

Link	Length (pallet lengths)
ga1	8
ga2	6
ga3	12
ga4	4
ga5	6
ga6	7
ha7	7
ha8	6
ha9	10
ha10	9

Table 4.3: First Central Layer Link Lengths

Workstation	Buffer Size (pallets)	Products Handled	Processing Times (seconds)
wsta1	5	0	20
		2	15
		3	20
		5	15
		6	20
		8	15
wsta2	10	0	40
		1	45
		3	40
		4	45
		6	40
		7	45
wsta3	20	1	25
		2	35
		4	25
		5	35
		7	25
		8	35

Table 4.4: First Central Layer Workstation Characteristics

Link	Length (pallet lengths)
gb1	9
gb2	5
gb3	8
gb4	3
gb5	4
gb6	6
hb7	15
hb8	9
hb9	14
hb10	6

Table 4.5: Second Central Layer Link Lengths

Workstation	Buffer Size (pallets)	Products Handled	Processing Times (seconds)
wstb1	10	0	15
		1	15
		2	15
		6	20
		7	20
		8	20
wstb2	20	0	35
		1	35
		2	35
		3	45
		4	45
		5	45
wstb3	30	3	25
		4	25
		5	25
		6	40
		7	40
		8	40

Table 4.6: Second Central Layer Workstation Characteristics

Link	Length (pallet lengths)
gc1	8
gc2	9
gc3	12
gc4	4
gc5	7
gc6	3
hc7	10
hc8	7
hc9	9
hc10	8

Table 4.7: Output Layer Link Lengths

4.4 Customer Order Assumptions

In this simulation, three baseline assumptions are made concerning the customer orders and their arrivals. First, the arrival of customer orders is modeled as a Poisson Process with adjustable mean interarrival time. The orders will be registered in the customer order queue at the beginning of every minute. Second, the size of the customer orders coming into the customer order queue are random with a uniform size distribution of between 1 and 100 products. Finally, any product selected at random from a typical customer order has an equal probability of being any product in the product family.

4.5 Scheduling Policies

The two scheduling policies to be described in this section differ in the way they assign workstations to pallets traversing the MPMS system. Both of these scheduling policies are supported by the simulation tool developed in Chapter 3, and the simulation programmer defined operations used to implement them may be found at the end of Appendix B. Before describing these scheduling policies, it is first necessary to discuss three fundamental scheduling issues. First, what criteria will be used to determine which output stream to assign to a particular customer order? Second, when loading a pallet onto an input stream, how is the decision made regarding which product type that pallet will carry, and which customer order will the product belong to? Finally, once a pallet is on an input stream, under what conditions should the pallet be discharged into the first central layer. The decision rules used for these scheduling issues will be the same for both scheduling policies.

When a customer order arrives, it is immediately placed in the customer order queue and assigned to an output stream. The order is assigned to the output stream which has the smallest number of undelivered products assigned to it. The term undelivered products to an output stream refers to the sum of all uninitiated products in the customer order queue belonging to orders assigned to the output stream, and pallets currently on the guideway system heading towards the output stream.

The input layer controller is continually searching for input streams which can accept a pallet. When an input stream is found, the layer controller attempts to load it with a pallet. The decision as to which part the pallet will carry is made as follows: The customer order queue is searched from its oldest to its most recent customer order for the first product type

which can be loaded onto the input stream. Within each customer order, the search begins by considering products in increasing order of product type number. For example, product 1 is considered before product 2. If a product is found, then a pallet for that product is loaded onto the input stream. Otherwise the input stream remains empty. Once a pallet is placed onto an input stream, it is discharged to the input stream's downstream link as soon as it becomes accessible.

4.5.1 Scheduling Policy A: Local Feedback Scheduler

Scheduling policy A is referred to as a local feedback scheduler because the information used in assigning a pallet's next workstation destination is restricted to the layer the pallet is currently in. Figure 4-2 illustrates how the local feedback scheduling policy works. After being discharged from the input stream, a pallet traverses the input stream's downstream link in the first central layer. Upon reaching the end of the input link, the pallet is then passed to the link's downstream junction. This junction, the first junction in the first central layer that the pallet has come in contact with, recognizes that the pallet needs directions. The junction then contacts the first central layer-controller, which selects a workstation in the first layer to assign to the pallet. The layer-controller chooses the workstation with the lowest associated cost $J(W_i)$ defined as follows:

$$J(W_i) = D + L_p + W_p + P_p$$

$J(W_i)$ is a rough estimate of how much time in seconds it will take for the pallet's task to be completed at workstation W_i. The cost equation takes four factors into account:

1. D = The sum of the lengths of the links the pallet must traverse to reach the workstation
2. L_p = The product of the workstation's mean processing time per task and the number of pallets in the link immediately upstream of the workstation.
3. W_p = The time the workstation would require to process all pallets currently in its buffer and machine if it could work continuously.
4. P_p = The time the workstation requires to process the pallet we are assigning a workstation to.

Along with assigning a workstation to the pallet, the layer-controller also assigns to

the pallet the sequence of links it must traverse to reach the workstation. The junction then uses the pallet's newly acquired information to move it to the appropriate downstream link. This information will also be used by other junctions encountered by the pallet while heading towards its assigned workstation. These junctions will use the pallet's information to direct the pallet to the appropriate downstream links.

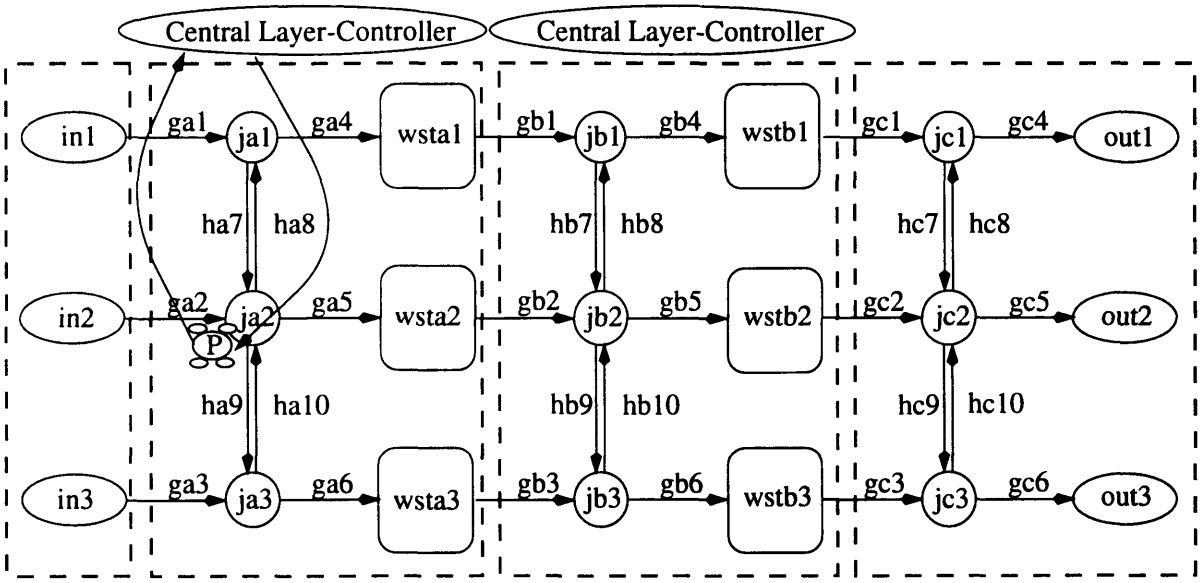

Figure 4-2: Simple MPMS System under Local Feedback Scheduling Policy

After being serviced by the workstation, the pallet is discharged to the workstation's downstream link in the second central layer. What happens next is essentially identical to what happened in the first central layer. Namely, upon encountering the first junction in the second layer, the junction informs the second central layer-controller that the pallet needs directions. Then, the second layer-controller chooses a workstation for the pallet by using the same cost function used by the first central layer controller.

4.5.2 Scheduling Policy B: Route Based Scheduler

Scheduling policy B is referred to as a route based scheduling policy because when a pallet needs directions, it is assigned a route through the entire system, not just a next workstation. Referring to Figure 4-3, the route based scheduler works as follows. After being discharged from an input stream, a pallet traverses the input stream's downstream link in the first central layer. Upon reaching the end of the input link, the pallet is then passed to the link's downstream junction, the first junction encountered by the pallet in the

entire system. This junction will be referred to as j_1. The junction, j_1, recognizes that the pallet needs directions.

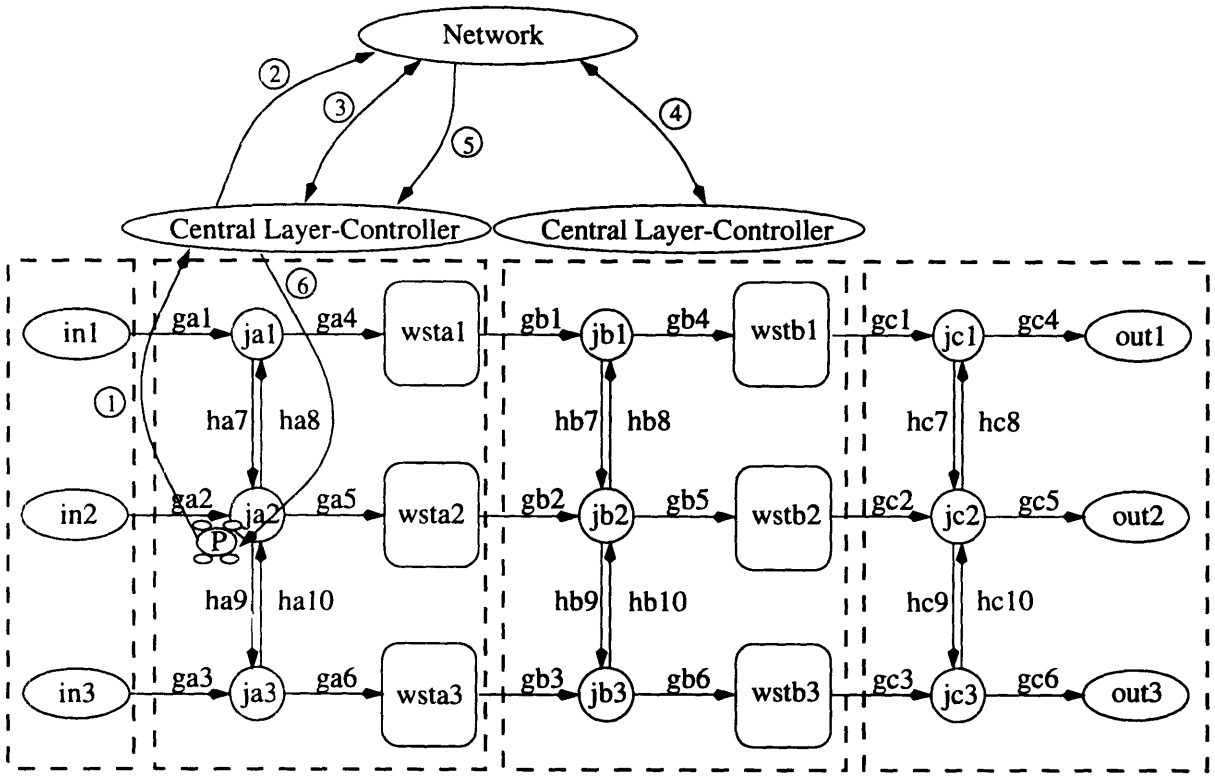

Figure 4-3: Simple MPMS System under Route Based Scheduling Policy

Step 1: The junction, j_1, contacts the first central layer-controller, informing it that one of its pallets needs directions. **Step 2**: The first central layer-controller then contacts the network, informing it that one of the pallets in one of its junctions needs directions. **Step 3**: The network then requests that the first central layer-controller determine which workstation in its layer to assign to the pallet, based on the local feedback scheduling policy just described. The first central layer-controller obliges the network's request, choosing a workstation, W_1. Next, the network determines the first junction in the second central layer a pallet moving downstream from W_1 would encounter. Call this junction j_2. **Step 4**: The network then requests that the second central layer-controller determine which workstation in its layer it would assign the pallet if it were at j_2, based on the local feedback scheduling policy. The second central layer-controller obliges the network's request, choosing a workstation, W_2. **Step 5**: The network passes the two workstation assignments; W_1, and W_2, to the first central layer-controller. **Step 6**: Finally, the first central layer-controller

assigns the workstations to the pallet. It also assigns to the pallet the sequence of links it must traverse to reach W_1.

Now that the pallet has been assigned a route through the system, junction j_1 uses the pallet's newly acquired information to move it to the appropriate downstream link. This information will also be used by other junctions encountered by the pallet while heading towards W_1. These junctions will use the pallet's information to direct the pallet to the appropriate downstream links. After being serviced by W_1, the pallet will be discharged to W_1's downstream link, and will eventually reach junction j_2. When the pallet reaches j_2, j_2 will inform the second central layer-controller that the pallet needs directions. The layer-controller will see that the pallet already has a workstation assignment, namely W_2. Because of this, it will assign to the pallet the sequence of links it must traverse to reach W_2. Junctions in the second central layer will then use the link assignments to route the pallet to W_2.

4.6 Comparison of Policies A and B

From the characteristics of the two scheduling policies described in the previous sections, it is expected that policy A, the local feedback scheduler will outperform policy B, the route based scheduler. This is because, the local feedback scheduler makes workstation assignments to pallets on a per central layer basis, whereas the route based scheduler assigns workstations for all the central layers at once. Hence, the local feedback scheduler makes its decisions using more current information on the state of the system than the route based scheduler. This result will be shown in Section 4.7.

In Section 4.7, the simple MPMS system will be simulated under the control of the two scheduling policies. Each simulation run will be performed for 480 hours (simulated time). In each simulation run, the mean size and the probabilistic distribution of customer orders will remain the same. The simulation runs will be grouped in pairs, each pair having the same mean customer order interarrival time and hence the same load or strain on the system. Furthermore, in each pair, the same "seed" will be used to generate the random customer order sequences. This will allow for a performance comparison of the two scheduling policies using the same sequence of customer orders. Finally, histograms will be given comparing the times each policy requires to process customer orders. Four pairs of histograms will be

made, each pair dedicated to a different mean customer order interarrival time.

4.7 Results

Histograms shown in Figures 4-4– 4-7 depict the results of the simulation runs. Although each simulation was run for 480 hours, the histograms only represent the latter 456 hours. The first 24 hours of the simulation data was not included in the histograms or mean order processing times so as to properly represent the steady state performance of the policies. The assumption that steady state performance is reached within the first 24 hours is based on the fact that the average processing time for a pallet is small relative to 24 hours. Hence, within the first 24 hours, the transient effect of pallets coming into an empty system is dissipated. The main program used to simulate the system under the local feedback scheduler, and generate the data used for the local feedback scheduler histogram in Figure 4-4 is given in Appendix C. This program, modified slightly, produces any simulation used below.

The first three pairs of histograms, Figures 4-4– 4-6, show that the distribution of times to complete orders is weighted more towards zero in scheduling policy A than in scheduling policy B. This results from the fact that orders were processed more quickly using scheduling policy A than scheduling policy B. These histograms are representative of the steady state performance of the two scheduling policies under increasingly demanding conditions. For each simulation run used to generate these histograms, the average rate of demand for products needed to fill customer orders is smaller than the average product processing rate of the system. The final histogram, Figure 4-7, shows the opposite scenario, one in which the average rate of demand for products is greater than the average product processing rate of the system. In this case, steady state performance cannot be reached and the time to process an order increases as the simulation time increases. This order processing time increase is due to order build-up in the customer order queue.

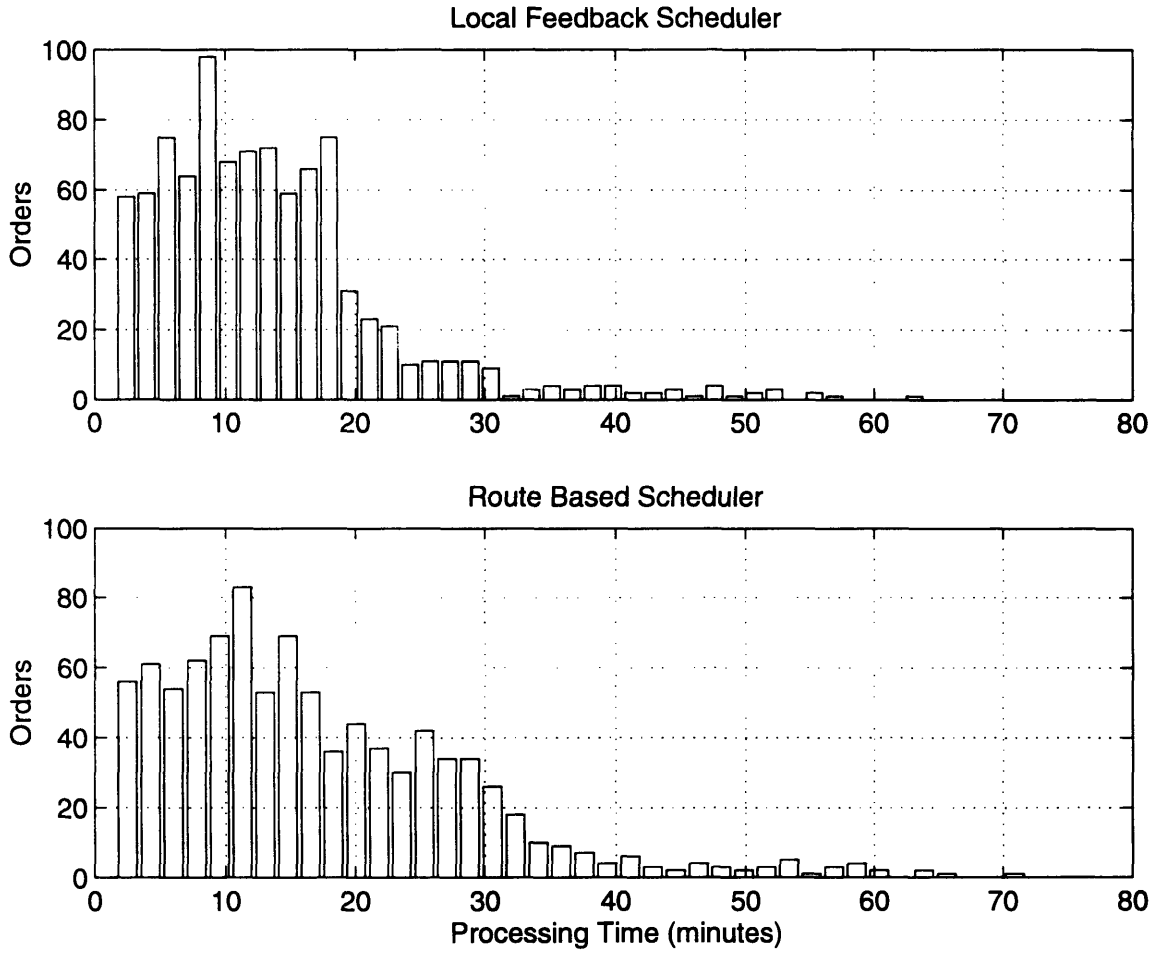

Figure 4-4: Histogram Comparison for $\lambda = 0.0333$

Figure 4-4 compares the two scheduling policies when the mean customer order interarrival rate is set to $\lambda = 0.0333$. This corresponds to an average of one customer order every 30 minutes. In 480 hours, a total of 986 customer orders were registered into the customer order queue. Both scheduling policies processed all of the customer orders in 480 hours. In the first 24 hours, both policies completed 54 orders. The histograms in Figure 4-4 show the processing time distributions of the remaining 932 orders. The local feedback scheduler and the route based scheduler had mean processing times per order of 13.57 and 17.17 minutes respectively.

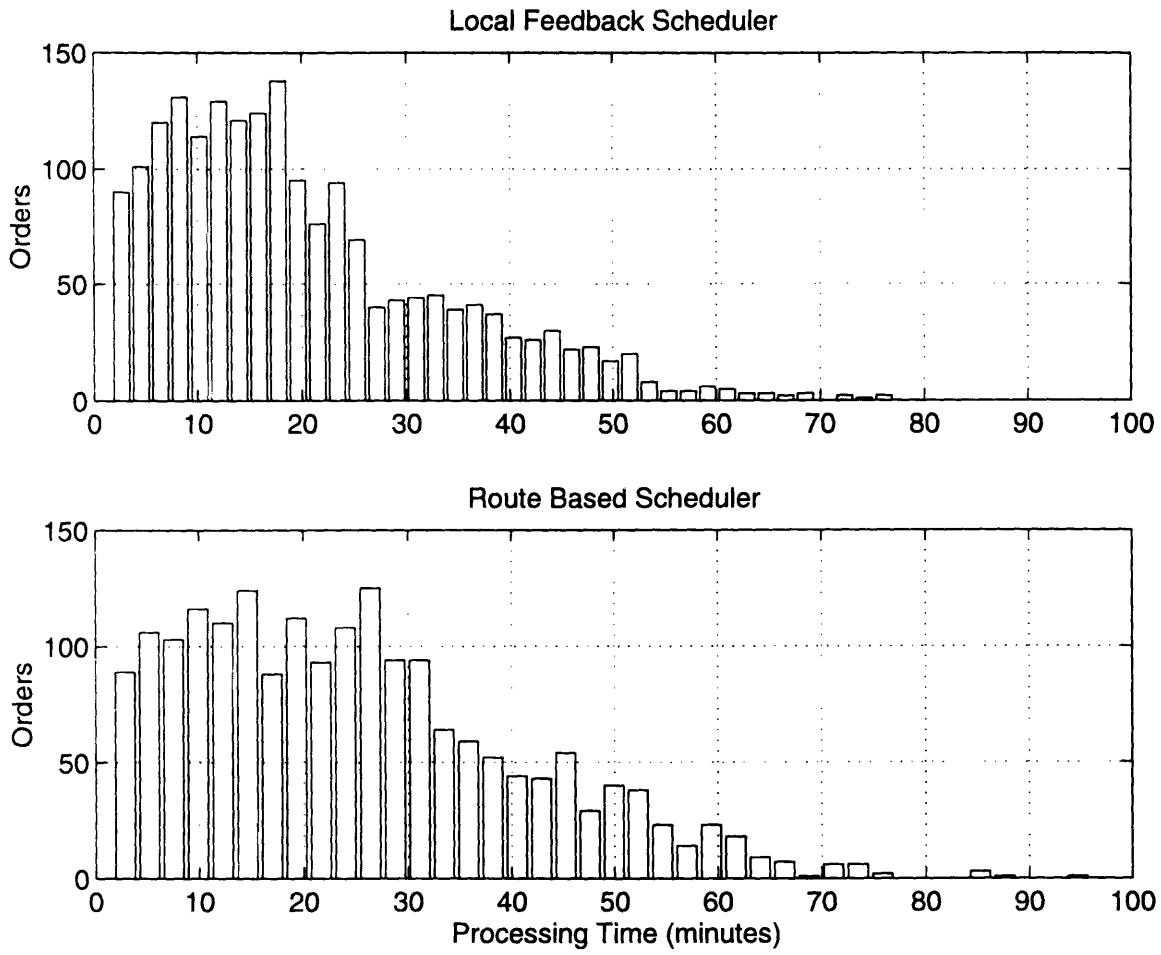

Figure 4-5: Histogram Comparison for $\lambda = 0.0667$

Figure 4-5 compares the two scheduling policies when the mean customer order interarrival rate is set to $\lambda = 0.0667$. This corresponds to an average of one customer order every 15 minutes. In 480 hours, a total of 1996 customer orders were registered into the customer order queue. Both scheduling policies processed all of the customer orders in 480 hours. In the first 24 hours, both policies completed 97 orders. The histograms in Figure 4-5 show the processing time distributions of the remaining 1899 orders. The local feedback scheduler and the route based scheduler had mean processing times per order of 20.22 and 25.21 minutes respectively.

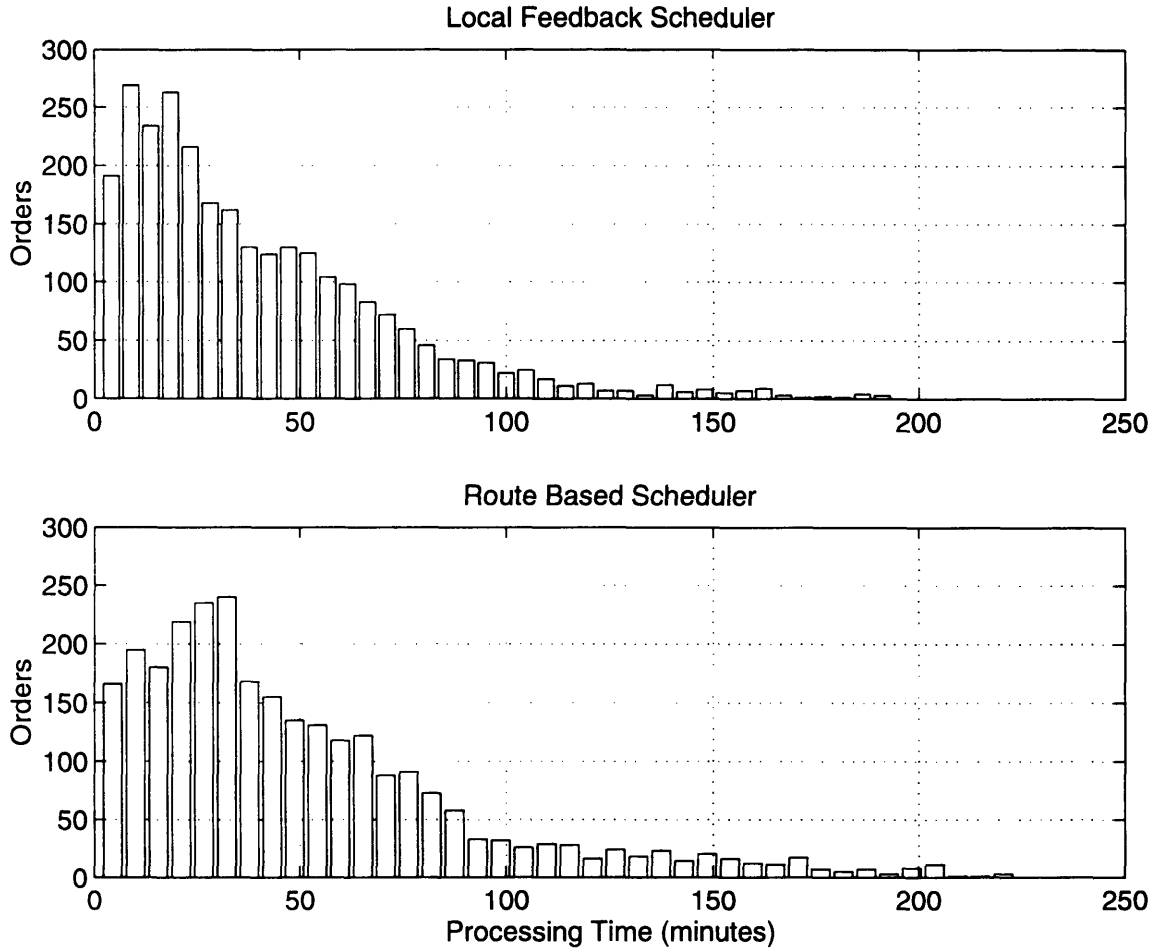

Figure 4-6: Histogram Comparison for $\lambda = 0.1000$

Figure 4-6 compares the two scheduling policies when the mean customer order interarrival rate is set to $\lambda = 0.1000$. This corresponds to an average of one customer order every 10 minutes. In 480 hours, a total of 2881 customer orders were registered into the customer order queue. Both scheduling policies processed all of the customer orders in 480 hours. In the first 24 hours, both policies completed 142 orders. The histograms in Figure 4-6 show the processing time distributions of the remaining 2739 orders. The local feedback scheduler and the route based scheduler had mean processing times per order of 40.08 and 50.73 minutes respectively.

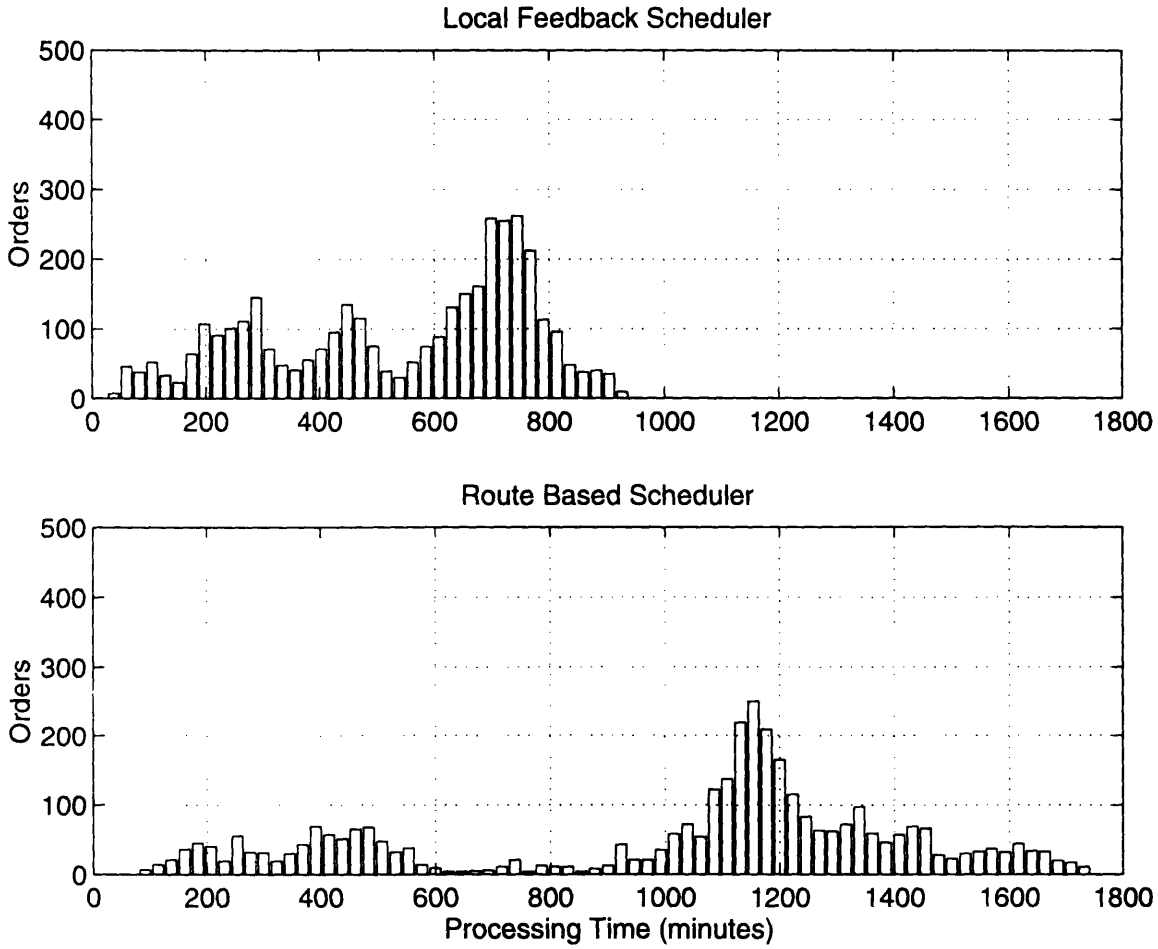

Figure 4-7: Histogram Comparison for $\lambda = 0.1333$

Figure 4-7 compares the two scheduling policies when the mean customer order interarrival rate is set to $\lambda = 0.1333$. This corresponds to an average of one customer order every 7.5 minutes. In 480 hours, a total of 3907 customer orders were registered into the customer order queue. Within the 480 hours, the local feedback scheduler processed only 3792 customer orders, while the route based scheduler processed even less, only 3671 customer orders. In the first 24 hours, scheduling policy A completed 178 orders and scheduling policy B completed 174 orders. The histograms in Figure 4-7 show the processing time distributions of the 3614 and 3497 orders processed by the local feedback scheduler and the route based scheduler in the remaining 456 hours, respectively. The local feedback scheduler and the route based scheduler had mean processing times per order of 539.75 and 1018.7 minutes respectively. Note that these mean processing times are not steady state values, as no steady state was reached for $\lambda = 0.1333$.

Chapter 5

Conclusion

5.1 Thesis Results

The objectives of this thesis were met in full. In Chapter 2, the MPMS class of flexible discrete part manufacturing systems was introduced. The class of MPMS systems was conceptualized as a general class of distributed cooperatively controlled flexible manufacturing systems. MPMS systems are described as the aggregate of a number of fundamental, self contained, intelligent system-level components. The behavior of these system-level components, as well as the communications architecture, common to all MPMS systems, were explained and modeled. Chapter 3 described the C++ implementation of the object-oriented tool for simulating MPMS systems. The simulation tool was designed so that at any time step, multiple system-level components may be performing operations independently of one another. Using this tool, any MPMS system can be constructed and simulated. To do this, one must first specify the characteristics of its constituent system-level components, and how they are interconnected. Second, in order to control the system, the simulation programmer specified functions must be defined. Hence, the simulation tool achieved the goal of capturing the open system qualities of the MPMS system.

In Chapter 4, a simple MPMS system model was quickly constructed using the simulation tool. Two different scheduling policies, referred to as the local feedback scheduler and the route based scheduler were then defined to control the system. The principal difference between these two policies is that the local feedback scheduler uses more current information to route pallets through the system. The simple system was then simulated under varying levels of production demand. The benefits of the simulation under the control of

both scheduling policies was twofold: First, successful simulation verified the correctness of the MPMS system model. That is, given enough time, customer orders and pallets were always processed correctly. Second, the results of the performance comparison between the two scheduling policies was in conformance with expectation. It was found that irrespective of the production demand placed on the system, the local feedback scheduling policy always outperformed the route based scheduling policy. Therefore, an argument can be made that the simulation tool may be used to test the relative performance of other, possibly less disparate scheduling policies.

5.2 Suggestions for Further Research

The model for the class of MPMS systems, and the simulation tool developed in this thesis lay the foundation for future research in scheduling MPMS systems. Room for improvements exists in both the model and the simulation tool. Richer and more diversified system-level component models can be introduced. To use the workstation as an example: As currently modeled, all workstations with buffers in an MPMS system model have the same buffer discipline. An improved model would allow for workstations with different buffer disciplines in any given MPMS system model. Also, a richer model would allow for such amenities as random workstation processing times, and account for change over times when switching from one task to another. Finally, for realism, failures such as system-level component breakdown or pallets carrying defective products can be incorporated into the MPMS system model. In addition to improving the MPMS system model, the simulations themselves can become more advanced. For example, more interesting customer order distributions, compositions, and arrival rates can be used to produce more realistic simulations. Finally, and in closing, there are also several generalizations which can be made to the class of MPMS systems itself, hence allowing for the inclusion of a wider range of flexible manufacturing systems. For example, the class can be generalized to allow for pallet cycling. That is, pallets would no longer be barred from reentering a layer to visit a workstation. Another improvement would be to allow for assembly operations in which pallets carrying product subassemblies meet in the system so that their subassemblies may be joined.

Appendix A

Random Order Generator - C Code (forder_comp.cc)

```
#include <stdlib.h>
#include <math.h>
#define PARTS  9

float ran1(long *idum);

struct ORDER{
        int ord_sz        ;
        int seq[PARTS]    ;
         };                                                    10



struct ORDER  forder_comp(int OMAX, long *cidum )
   {
       struct ORDER order;

       order.ord_sz = 0;                  /* initialize order */
       for(int i=0; i<PARTS; i++){
         order.seq[i] = 0 ;}                                   20

       float tmp;
       int k;
```

```
    do{ tmp = ((OMAX)*ran1(cidum)+1) ;  /* Random generation of order size */
    order.ord_sz= (int) floor(tmp);} while(order.ord_sz==(OMAX+1));
int ord_sz = order.ord_sz;


for(int i=1; i<= ord_sz; i++){  /* Random generation of order composition */
  do{
     tmp = (PARTS)*ran1(cidum);
     k   = (int) floor(tmp);} while( k == ord_sz);
  order.seq[k] += 1;}
    return order;}
```

Appendix B

System Header Files - C++

B.1 IN_QUEUE and Time (SECmyhead.h)

```
#define PARTS 9

struct ORDER{
        int ord_sz     ;
        int seq[PARTS] ;};

extern class Time t;

//*************************************************************************************
//*******
//*******      TIME OBJECT
//*******

struct TCK{
  int day ;
  int hour;
  int minute;
  int second;};

class Time
{
private:
 int day;
```

```
 int hour;   //   military time
 int minute;
 int second;
public:
 Time();                        //   sets time to day:hour:min:sec= 1:0:0:0
 Time(int d, int h, int m, int s): //   sets time to day:hour:min:sec= d:h:m:s
 Time(TCK tick);                //   sets time to tick.day:tick.hour: .. etc.

 ~Time(){}

 Time(const Time &);
 Time & operator=(const Time &);
 friend Time operator+( Time & L, Time & R);

 void inc();
 TCK ck() const;
};
 Time::Time(): day(1), hour(0), minute(0), second(0) {}
 Time::Time(int d, int h, int m, int s): day(d), hour(h), minute(m), second(s) {}
 Time::Time(TCK tick):day(tick.day), hour(tick.hour), minute(tick.minute),
                second(tick.second){}

 Time::Time(const Time & S){  //   this copy constructor is probably the
     day    = S.day;        //   default used by the compiler
     hour   = S.hour;
     minute = S.minute;
     second = S.second;}

 Time & Time::operator=(const Time & S){
     if( this != &S){
       day    = S.day;        //   same coment for assignment.
       hour   = S.hour;
       minute = S.minute;
       second = S.second;}
     return *this;}

 Time operator+(Time & L, Time & R){
    Time temp;
    temp.second = L.second + R.second;
```

```
    temp.minute = (temp.second/60) + L.minute + R.minute;
                temp.second = temp.second%60;
    temp.hour   = (temp.minute/60) + L.hour + R.hour;
                temp.minute = temp.minute%60;
    temp.day    = (temp.hour/24) + L.day + R.day;
                temp.hour = temp.hour%24;
    return temp;}                                                          70

void Time::inc(){
    second +=1;
    minute +=(second/60);
           second = second%60;
    hour   +=(minute/60);
           minute = minute%60;
    day    +=(hour/24);
           hour = hour%24;}
                                                                           80
TCK Time::ck() const{
    TCK tmp={day, hour, minute, second};
    return tmp;}

//*****************************************************************************
//******
//******      IN_ORDER Order object stored in the Order_Q
//******


                                                                           90
//*****
//*****       COMPOSITION, the component class of IN_ORDER
//*****

class COMP;
class OUT ;

class COMPOSITION
{
private:                                                                  100
 int *order_point ;
```

```
public:
 COMPOSITION();
 COMPOSITION(int *);
~COMPOSITION() { delete [] order_point;}
 COMPOSITION(const COMPOSITION &);
 COMPOSITION & operator=(const COMPOSITION &);

// Interface Functions                                                    110
int  comp(int PART) const:      // returns the number of PART in the order
void comp(int PART, int CHANGE); // modifies the order by changing the number
                          // PARTs by CHANGE
};

COMPOSITION::COMPOSITION(){       // default COMPOSITION constructor
      order_point = new int[PARTS]; // COMPOSITION object initialized to 0
      for(int i=0; i<PARTS ; i++) order_point[i] = 0;}

COMPOSITION::COMPOSITION(int * source){                                   120
      order_point = new int[PARTS];
      for(int i=0; i<PARTS ; i++) (*(order_point + i)) = (*(source + i));}

COMPOSITION::COMPOSITION(const COMPOSITION & C){
      order_point = new int[PARTS];
      for(int i=0; i<PARTS ; i++) (*(order_point +i)) = (*(C.order_point +i));}

COMPOSITION &  COMPOSITION::operator=(const COMPOSITION & C){
      if ( this != &C){
                                                                          130
      delete [] order_point;        // note that because the length of
      order_point = new int[PARTS]; // order_point[] is fixed, these two steps
                              // are really unnecessary, I included them
                              // as an exercise.

      for(int i=0; i<PARTS ; i++) (*(order_point +i)) = (*(C.order_point +i));}
      return  *this;}
int  COMPOSITION::comp(int PART) const { return *(order_point + PART);}
void COMPOSITION::comp(int PART, int CHANGE) {*(order_point + PART) += CHANGE;}
                                                                          140
//******
```

```
//******      IN_ORDER
//******

class IN_ORDER
{
private:
 int   order_id;
 TCK   date_issued;
 const int order_sz  ;                                                    150
     int present_sz;

 const COMPOSITION order_comp  ;
     COMPOSITION present_comp;

 TCK    due_date;
 OUT * out;

 TCK   due_date_assign();    //   sets order's due date, note that this is private
                                                                          160
 IN_ORDER(const IN_ORDER &);              //   copy constructor, not implemented
 IN_ORDER & operator=(const IN_ORDER &);  //   assignment, not implemented


 friend class IN_QUEUE;

 IN_ORDER * previous;      //   pointers used by ORDER_Q to fascilitate
 IN_ORDER * next    ;      //   the doublely linked list

public:                                                                   170
 IN_ORDER(struct ORDER, IN_ORDER * n, IN_ORDER * p);    //   constructor
 ~IN_ORDER(){}                                  //   destructor

 //  Interface Fucntions

 int   id() const;                //   returns order's id number
 TCK   ck_issue() const;          //   returns order's issued date
 TCK   ck_due() const  ;          //   returns order's due date
 int   o_size() const;            //   returns order's original size
 int   r_size() const;            //   returns size of un-initiated order    180
```

```
int   o_comp(int PART) const;        //   returns number of PART in orig. order
int   r_comp(int PART) const;        //   returns number of PART in remaining
                                     //   un-initiated order
int change_comp(int PART, int CHANGE);//   changes the number of PART in order
                                     //   by CHANGE
COMPOSITION original_c() const;      //   returns IN_ORDER's order_comp
void   out_stream(OUT *);            //   sets the output destination of order
OUT *  out_stream()     ;            //rreturns the output destination of order
};
                                                                              190
IN_ORDER::IN_ORDER(struct ORDER s, IN_ORDER * p, IN_ORDER * n) : order_sz(s.ord_sz),
     order_comp(s.seq), previous(p), next(n), out(0), order_id(0){
     present_sz   = order_sz;        //  due_date and date_issued are assigned
     present_comp = order_comp ;     //  upon IN_ORDER creation.
     date_issued  = t.ck();
     due_date     = due_date_assign();}

TCK   IN_ORDER::due_date_assign(){   //   current Time is an implicit argument.
     Time t1(0,0,15,0)  ;
     Time due= t + t1;       //   due_date one hour after IN_ORDER is assigned      200
     return due.ck()   ;}

TCK   IN_ORDER::ck_issue() const {return date_issued ;}
TCK   IN_ORDER::ck_due() const {return due_date;}
int   IN_ORDER::o_size() const {return order_sz;    }
int   IN_ORDER::r_size() const {return present_sz;  }
int   IN_ORDER::o_comp(int PART) const { return order_comp.comp(PART)  ;}
int   IN_ORDER::r_comp(int PART) const { return present_comp.comp(PART);}
int   IN_ORDER::change_comp(int PART, int CHANGE){
     present_comp.comp(PART,CHANGE);                                          210
     present_sz += CHANGE;
     if(present_sz==0) return 1;  //   Return 1 to indicate that IN_ORDER is empty
     return 0;}
int   IN_ORDER::id() const { return order_id;}

COMPOSITION IN_ORDER::original_c() const { return order_comp;}

void   IN_ORDER::out_stream(OUT * o_stream){out = o_stream;}
```

```
OUT * IN_ORDER::out_stream(){return out;}                                    220

//*****************************************************************************
//******
//******                    Class IN_QUEUE
//******
//*****************************************************************************
class IN_QUEUE
{
private:
 int order_id;                                                               230
 IN_ORDER * c;     //   iterator currency
 IN_ORDER * lead;  //   newest object in list
 IN_ORDER * rear:  //   oldest object in list

//RRepresentational Invariant:
//      each IN_ORDER has a unique order_id, lead always points to the
//      newest IN_ORDER in the IN_QUEUE, c always addresses a valid IN_ORDER
//      or zero, all IN_QUEUE elements have next and previous pointing to valid
//      IN_ORDER objects with the exeptions: the oldest IN_ORDER has next=0, and
//      the newest has previous=0.                                            240



 IN_QUEUE( const IN_QUEUE &);          //   copy constructor, not implemented
 IN_QUEUE & operator=(const IN_QUEUE); //   assignment, not implemented

public:
 IN_QUEUE();
 ~IN_QUEUE();
                                                                             250

 int  insert(struct ORDER);    //   inserts order into Order_Q, return order_id
 int  remove(int order_id);    //   removes order_id from ORDER_Q, return order_id
 void reset_newest();          //   resets the currency to newest Queue element
 void reset_oldest();          //   resets the currency to the oldest Queue element
 IN_ORDER * isIn(int order_id); //   returns * to order_id if it is in Queue, else 0
 IN_ORDER * next();            //   returns the current order pointer, and increments
                               //   the currency, or returns zero if no more orders
```

```
 IN_ORDER * prev();          //  returns the current order pointer, and decrements
                             //  the currency, or returns zero if no more orders

};

IN_QUEUE::IN_QUEUE(): c(0), lead(0), rear(0), order_id(1) {}

IN_QUEUE::~IN_QUEUE(){
      if(lead) cout << "Terminating IN_QUEUE not empty\n";
      while(lead != 0){
        IN_ORDER *t = lead->next;
        delete lead;
        lead = t;}}


int IN_QUEUE::insert(struct ORDER s){
      IN_ORDER * temp = lead;
      lead = new IN_ORDER(s,0,lead); //  create new IN_ORDER with next link
      lead->order_id=(order_id++);  //  set order_id number
      if(temp) temp->previous=lead; //  if second newest object exists, link it
                                    //  to newest object
      else c=rear=lead;             //  line only executes if IN_QUEUE was empty
      return lead->order_id;}

int IN_QUEUE::remove(int order_id){
      IN_ORDER * curr = lead ;
      IN_ORDER * prev = 0    ;
      if(!lead) cout << "No orders to remove, Queue is empty.\n";
      while(curr){
        if (curr->order_id == order_id){
          if(prev){                 //  removing any order but the newest
            prev->next = curr->next; //  link from newer to older
            if(prev->next)          //  if older exists, link older to newer
              prev->next->previous = prev;
            else  rear=prev;}       //  order removing is rear; assign new rear
          else{                     //  removing newest order
            lead = curr->next;      //  set lead equal to newest order
            if(lead)                //  if queue is not empty, set the
              lead->previous =0;    //  lead->previous order to 0.
```

```
        else cout << "IN_QUEUE is now empty\n";}
      if(c==curr) c=curr->next; //  IMPORTANT:  KEEPS c->next VALID
      delete curr;              //  IF c==curr AND CURR IS DELETED                300
      return order_id;}
    prev = curr;
    curr = curr->next;}
  cout << "Order was not in Queue.\n";
  return 0;}                    //  returns 0 if order_id does not exist

void IN_QUEUE::reset_newest(){ c = lead;}
void IN_QUEUE::reset_oldest(){ c = rear;}

IN_ORDER * IN_QUEUE::next(){                                                    310
  if(c){
    IN_ORDER * r = c;
    c=c->next;
    return r;}
  else
    return 0;}

IN_ORDER * IN_QUEUE::prev(){
  if(c){
    IN_ORDER * r = c;                                                           320
    c=c->previous;
    return r;}
  else
    return 0;}

IN_ORDER * IN_QUEUE::isIn(int order_id){   //  search starts from newest element
  IN_ORDER * q;                    //  Queue
  reset_newest();
  while( (q=next()) != 0 ){
    if(q->order_id == order_id)                                                 330
      return q;}
  cout << "ORDER #"<<order_id<< " is not in Queue\n";
  return 0;}
```

B.2 MPMS System Implementation (SECLAYERS.h)

```
class COMP  ; //   COMPonent base class
class IN    ; //   IN,          object = 0
class LNK   ; //   LINK,        object = 1
class JCT   ; //   JUNCTION,    object = 2
class WST   ; //   WORKSTATION, object = 3
class OUT   ; //   OUT          , object = 4

class Input_Layer  ;
class Central_Layer;
class Output_Layer ;                                                    10

extern class IN_QUEUE Q;
extern class Time     t;

struct typ{
     int layer ;  //   layer specifies where the objcet is found. object
     int object;  //   specifies the type of object see class definitions.
     int id_num;} //   id_num is a unique identifier associated with
                  //   an object in a layer
     initial={0,0,0};                                                   20

//*****************************************************************************

class PATH;

class PALLET
{
 private:
  friend class OUT;
                                                                        30
  int can_del_flag     ;  //   1 = Pallet may be deleted, 0 = may not
  void can_delete()    ;  //   sets can_del_flag = to 1

  int  dir_len         ;  //   number of LNK *'s in direction list
  int  dir_ind         ;  //   direction list index
  LNK ** Direction     ;  //   direction list
```

```
int  wst_len        ;  //   number of WST *'s in workstation target list
int  wst_ind        ;  //   workstation target list index
WST ** WST_target   ;  //   workstation target list                          40

int  d_need         ;  //   need direction flag: 1 = need, 0 = don't need
int  w_need         ;  //   need workstation flag: 1 = need, 0= don't need
LNK * last_lnk      ;  //   last link to tick PALLET

const int order     ;  //   order pallet belongs to, defined only in constructor
const int part      ;  //   part type of pallet    , defined only in constructor
OUT * const out     ;  //   pointer to pallet's OUT, defined only in constructor

const TCK du        ;  //   due date of order pallet belongs to              50
int      pr         ;  //   priority of a specific pallet

~PALLET(){if(can_del_flag==0) cerr<<"Incorrectly trying to delete PALLET";}

PALLET(const PALLET &);              //  copy constructor, not implemented
PALLET & operator=(const PALLET &);  //  assignment, not implemented

public:
PALLET(int ord, int par, TCK d, OUT * ot): order(ord), part(par), dir_len(0),
            dir_ind(0), wst_len(0), wst_ind(0) , d_need(1) , w_need(1),      60
            out(ot)   , Direction(0), WST_target(0), du(d), pr(0),
            can_del_flag(0),last_lnk(0){}

LNK *   next_link() ;      //   return pointer to next LNK
WST *   next_wst()  ;      //   return pointer to next target WST

int   dir_need()     ;   //   does PALLET need new directions  to a WORKSTATION
int   wst_need()     ;   //   does PALLET need a new target WORSTATION

OUT * out_ptr() const  ;   //   return pointer to pallet's OUT               70
int   order_num() const ;  //   return order # pallet belongs to
int   part_num() const  ;  //   part type of pallet

TCK   ck_due() const   ;   //   return PALLET due date
void  priority(int)    ;   //   set PALLET priority
int   priority()       ;   //   return PALLET priority
```

```
 void tick(LNK *)      ;    //  update Direction and WST_target indexes

 void fix_direc(int, LNK **) ;//  fix direction list                                    80
 void fix_direc(int, WST **) ;//  fix workstation target

 LNK * last_tick_on()   ;    //  return last link to tick PALLET
};

void PALLET::can_delete(){ can_del_flag= 1;}

LNK * PALLET::next_link(){if(dir_ind < dir_len) return Direction[dir_ind];
      else return 0;}                                          //  error
WST * PALLET::next_wst(){ if(wst_ind < wst_len) return WST_target[wst_ind];          90
      else {cerr <<"error @ PALLET::next_wst()\n"; return 0;}}        //  error

int  PALLET::dir_need(){ if(dir_len==0) return 1; else return 0;}
int  PALLET::wst_need(){ if(wst_len==0) return 1; else return 0;}

OUT * PALLET::out_ptr() const { return out;}
int   PALLET::order_num() const{ return order;}
int   PALLET::part_num() const{ return part;}

TCK   PALLET::ck_due() const{ return du;}                                            100
void  PALLET::priority(int p){ pr = p;}
int   PALLET::priority(){ return pr;}


void PALLET::tick(LNK * lnk){//   TICK tells pallet the link it was just on
     if(lnk) last_lnk = lnk;
     if(dir_ind < dir_len){//   all other changes only if PALLET has directions
       dir_ind += 1;                 //   Ticked at end of LNK
       if(dir_ind == dir_len){         //   if Directions are done
         dir_ind = 0; dir_len = 0;    //   raise the need_dir                        110
         free(Direction);            //   flag, increment wst_ind
         wst_ind +=1;
         if(wst_ind == wst_len){      //   if WST directions are
           wst_ind = 0; wst_len = 0; //   done, raise the need_wst
           free(WST_target);}}}}     //   flag
```

```
void PALLET::fix_direc(int size, LNK ** lnk){
                dir_len = size ;
                Direction = (LNK **)calloc(size,sizeof(LNK*));
                for(int i=0; i< size; i++) Direction[i] =lnk[i];}                    120

void PALLET::fix_direc(int size, WST ** wst){
                wst_len = size;
                WST_target = (WST**)calloc(size,sizeof(WST*));
                for(int i=0; i< size; i++) WST_target[i] =wst[i];}

LNK * PALLET::last_tick_on(){return last_lnk;}

//*****************************************************************************
                                                                                 130
class COMP
{
public:
     typ comp_typ;
  virtual void set_up(COMP *)=0          ; //  link to upstream object
  virtual void set_dw(COMP *)=0          ; //  link to downstream object
  virtual COMP * upstream()=0            ; //  return upstream COMPonent *
  virtual COMP * downstream()=0          ; //  return downstream COMPonent *
  virtual typ  COMP_type(int l, int id)=0 ; //  set comp_typ
       typ  C_type(){return comp_typ;}   //  return comp_typ                       140
  virtual int  access(COMP *)=0          ; //  return 1 = accessible, 0 not
  virtual void move(COMP *, PALLET *)=0  ; //  pass PALLET from upstream COMP

  COMP(){comp_typ.layer=0; comp_typ.object= 0; comp_typ.id_num=0;}
  virtual ~COMP(){};

//*****************************************************************************

//*** COMPonent to Layer Comunication handeled through abstract base class Layer:
                                                                                 150
class Layer
{
 protected:
  Layer(){}
```

```
public:
 virtual void inform(JCT *, PALLET *){cerr<<"invalid use of Layer::inform(J,P)";}
 virtual void done(OUT *, int){cerr<<"invalid use of Layer::inform(int)";}
 virtual ~Layer(){}};

//*** NOTE:  Output_Layer::done(int id) is called when an OUT has completed an order.
//***       This function embodies the action taken by the Output_Layer;
//***       it includes deleting the order from IN_QUEUE, as well as reporting
//***       the order completed to the POLICY COMPARISON functions

//*****************************************************************************

//*** Central_Layer to network communication handeled through base class Basenet:

class W_PATH;

class Basenet
{
 protected:
  Basenet(){}
  virtual ~Basenet(){cerr <<"deleting Basenet\n";}
 public:
  virtual W_PATH centralized_1(Central_Layer *cent_lay, JCT *jct, PALLET *pall)=0;};

//*****************************************************************************

class IN_base : public COMP       //  IN COMPonent
{
protected:
  COMP * up   ;                    //  pointer to upstream object
  COMP * down  ;                   //  pointer to downstream object
  virtual void set_up(COMP * U){ up =  U;} //  set U to 0  v1
  virtual void set_dw(COMP * D){ down = D;}          //  v2
  virtual typ COMP_type(int l, int id){              //  v5
        comp_typ.layer=l  ;
        comp_typ.object=0 ;
        comp_typ.id_num=id;
        return comp_typ;}
public :
```

```
 IN_base(int l, int id):COMP(),up(0),down(0){COMP_type(l,id);}
 ~IN_base(){} //  Durring the entire simulation LNK will be linked

 virtual COMP * upstream()  {return up  ;}            //  v3
 virtual COMP * downstream(){return down;}             //  v4
                                                                                    200
// join management interface
 friend void join(COMP & UPstream, COMP & DOwnstream);};//  joins two COMPonents

//*****************************************************************************

class LNK_base : public COMP        //  LINK COMPonent
{
protected:
 COMP * up    ;                     //  pointer to upstream object
 COMP * down  ;                     //  pointer to downstream object                 210
 virtual void set_up(COMP * U){ up =  U;}          //  v1
 virtual void set_dw(COMP * D){ down = D;}          //  v2
 virtual typ COMP_type(int l, int id){             //  v5
        comp_typ.layer=l  ;
        comp_typ.object=1 ;
        comp_typ.id_num=id;
        return comp_typ;}

 LNK_base(int l, int id):COMP(),up(0),down(0){COMP_type(l,id);}
 ~LNK_base(){}                                                                      220

public:
 virtual COMP * upstream()  {return up  ;}          //  v3
 virtual COMP * downstream(){return down;}           //  v4

// join management interface
 friend void join(COMP & UPstream, COMP & DOwnstream);};//  joins two COMPonents

//*****************************************************************************
                                                                                    230
class JCT_UP_LINK
{
```

```
 private:
  friend class UP_JCT     ;
  friend class JCT        ;
  JCT_UP_LINK * next      ;
  COMP     * dat          ;
  PALLET   * pallet       ;

  JCT_UP_LINK(COMP * d, JCT_UP_LINK * n):dat(d), next(n), pallet(0){}                240
 public:
  COMP   * data(){return dat;}
  PALLET * store(){return pallet;}
};

//*****************************************************************************************

class JCT_DW_LINK
{
 private:                                                                           250
  friend class DW_JCT     ;
  friend class JCT        ;
  JCT_DW_LINK * next      ;
  COMP     * dat          ;

  JCT_DW_LINK(COMP * d, JCT_DW_LINK * n):dat(d), next(n){}
 public:
  COMP   * data(){return dat;}
};
                                                                                    260
//*****************************************************************************************

class UP_JCT
{
 private:

  JCT_UP_LINK * lead  ;       //   points to head of list
  JCT_UP_LINK * c     ;       //   iterator currency
  JCT_UP_LINK * sw;           //   switching currency
                                                                                    270
public:
```

```
 JCT_UP_LINK * insert(COMP *)  ;//  insert COMPonent into UP_JCT, return link *
 void      reset()          ;//  reset c to lead link in UP_JCT
 JCT_UP_LINK * get()        ;//  returns first link * in UP_JCT
 JCT_UP_LINK * next()       ;//  returns current link * and increments c


 JCT_UP_LINK * next_sw()           ;//  cycle through UP_JCT and return COMP *'s
 void       reset_sw()          ;//  reset sw to lead component                    280
 PALLET     * isIn_p(COMP *)    ;//  return PALLET * in link given by COMP *
 void       putIn_p(COMP *,PALLET *) ;//  put PALLET * in link given by COMP *

 UP_JCT():c(0), lead(0), sw(0){}
 ~UP_JCT();
};

UP_JCT::~UP_JCT(){
     while(lead !=0){
       JCT_UP_LINK *t = lead->next;                                              290
       delete lead->dat;      //  delete upstream COMPonent
       delete lead;           //  delete link referencing the deleted COMPonent
       lead = t;}}

JCT_UP_LINK * UP_JCT::insert(COMP * d){
 JCT_UP_LINK * temp = lead;
 lead = new JCT_UP_LINK(d, lead);
 if(temp == 0){c= lead;} //  this line only executes if UP_JCT was empty
 return lead;}
                                                                                300
void  UP_JCT::reset(){c=lead;}

JCT_UP_LINK * UP_JCT::get(){
 if(lead)
  return lead;
 else
  return 0;}

JCT_UP_LINK * UP_JCT::next(){          //  cycles through UP_JCT
 if(lead){   //  if UP_JCT is not empty                                          310
```

```
  if(c){
     JCT_UP_LINK * r = c;
     c=c->next;
     return r;}
  else{
     reset();
     c=lead->next;
     return lead;}}
  return 0;}  //  no elements in UP_JCT
                                                                              320
JCT_UP_LINK * UP_JCT::next_sw(){        //  cycles through UP_JCT
 if(lead){   //  if JCT_UP_LINK is not empty  (and it should not be)
  if(sw){
     JCT_UP_LINK * r = sw;
     sw=sw->next;
     return r;}
  else{
     reset_sw();
     sw=lead->next;
     return lead;}}                                                           330
  return 0;}  //  no elements in JCT_list

void  UP_JCT::reset_sw(){sw=lead;}

PALLET * UP_JCT::isIn_p(COMP * U){   //  asume COMPonent U is in UP_JCT
        reset();
        while(c != 0){
          if(c->dat == U) return c->pallet;
          c=c->next;}}
                                                                              340
void    UP_JCT::putIn_p(COMP * U, PALLET * P){  //  assume COMPonent is in UP_JCT
        reset();
        while(c != 0){
          if(c->dat == U) c->pallet = P;
          c=c->next;}}

//*****************************************************************************

class DW_JCT
```

```
{                                                                          350
private:
 JCT_DW_LINK * lead  ;      //  points to head of list
 JCT_DW_LINK * c     ;      //  iterator currency

public:

 JCT_DW_LINK * insert(COMP *)    ;//  insert COMPonent into DW_JCT, return link *
 void         reset()          ;//  reset c to lead link in DW_JCT
 JCT_DW_LINK * get()           ;//  returns first link * in DW_JCT
 JCT_DW_LINK * next()          ;//  returns current link * and increments c      360
 COMP        * isIn_c(struct typ) ;//  return * to COMP identified by typ

 DW_JCT():lead(0), c(0){}
 ~DW_JCT();
};

DW_JCT::~DW_JCT(){
      while(lead !=0){
        JCT_DW_LINK *t = lead->next;
        delete lead->dat;      //  delete downstream COMPonent                 370
        delete lead;           //  delete link referencing the deleted COMPonent
        lead = t;}}

JCT_DW_LINK * DW_JCT::insert(COMP * d){
 JCT_DW_LINK * temp = lead;
 lead = new JCT_DW_LINK(d, lead);
 if(temp == 0){c= lead;} //  this line only executes if DW_JCT was empty
 return lead;}

void  DW_JCT::reset(){c=lead;}                                             380

JCT_DW_LINK * DW_JCT::get(){
 if(lead)
  return lead;
 else
  return 0;}

JCT_DW_LINK * DW_JCT::next(){          //  cycles through JCT_list
```

```
if(lead){    // if DW_JCT is not empty
  if(c){                                                                    390
      JCT_DW_LINK * r = c;
      c=c->next;
      return r;}
  else{
      reset();
      c=lead->next;
      return lead;}}
  return 0;}  // no elements in DW_JCT

COMP * DW_JCT::isIn_c(struct typ T){  //  assume COMPonent type T is in DW_JCT   400
       typ tmp;
       reset();
       while(c != 0){
         tmp= c->dat->C_type();
         if((tmp.id_num==T.id_num)&&(tmp.object==T.object)&&(tmp.layer==T.layer))
         return c->dat;
         c=c->next;}}

//*****************************************************************************
                                                                            410
class JCT_base : public COMP        //  JCT COMPonent
{
protected:
  UP_JCT  Up_List;                  //  pointers to upstream objects
  DW_JCT  Dw_List;                  //  pointers to downstream objects
  virtual void set_up(COMP * U);                    //  v1
  virtual void set_dw(COMP * D);                    //  v2
  virtual typ COMP_type(int l, int id){             //  v5
         comp_typ.layer=l  ;
         comp_typ.object=2 ;                                                420
         comp_typ.id_num=id;
         return comp_typ;}
public :
  Layer * L           ; //  Pointer to JCT's Layer
  JCT_base(int l, int id):COMP(), L(0) {COMP_type(l,id);}
  ~JCT_base(){} //  Durring the entire simulation JCT will be linked
```

```
 virtual COMP * upstream();                    //  v3
 virtual COMP * downstream();                  //  v4
                                                                              430
 virtual int  access(COMP *);                  //  v6
 virtual void move(COMP *, PALLET *);          //  v7

// join management interface
 friend void join(COMP & UPstream, COMP & DOwnstream);};//  joins two COMPonents


  void JCT_base::set_up(COMP * U){Up_List.insert(U);}

  void JCT_base::set_dw(COMP * D){Dw_List.insert(D);}                         440

  COMP * JCT_base::upstream(){//  cycle through Up_list elements
       return Up_List.next()->data();}

  COMP * JCT_base::downstream(){//  cycle through Dw_list elements
       return Dw_List.next()->data();}

  int JCT_base::access(COMP * U){
      if(Up_List.isIn_p(U)) return 0;
      else return 1;}                                                         450

  void JCT_base::move(COMP * U, PALLET * P){  //  assume access is available
      if(P->dir_need()){          //  if PALLET needs directions,
        L->inform((JCT *) this, P); //  inform Layer_Controler, get
        P->tick(0);}              //  instructions and tick PALLET
      Up_List.putIn_p(U,P);}

//*****************************************************************************

class WST_base : public COMP          //  WORKSTATION  COMPonent                460
{
protected:
 COMP * up   ;                        //  pointer to upstream object
 COMP * down ;                        //  pointer to downstream object
 virtual void set_up(COMP * U){ up =  U;}          //  v1
 virtual void set_dw(COMP * D){ down = D;}         //  v2
```

```
 virtual typ COMP_type(int l, int id){              //  v5
        comp_typ.layer=l  ;
        comp_typ.object=3 ;
        comp_typ.id_num=id;                                                    470
        return comp_typ;}
public :
 WST_base(int l, int id):COMP(),up(0),down(0){COMP_type(l,id);}
 ~WST_base(){} //  Durring the entire simulation WORKSTATION will be linked

 virtual COMP * upstream()  {return up  ;}          //  v3
 virtual COMP * downstream(){return down;}           //  v4

//  join management interface
 friend void join(COMP & UPstream, COMP & DOwnstream);};//  joins two COMPonents  480

//*****************************************************************************

class OUT_base : public COMP       //  OUT COMPonent
{
private:
 COMP * up    ;                    //  pointer to upstream object
 COMP * down  ;                    //  pointer to downstream object
 virtual void set_up(COMP * U){ up =  U;}          //  v1
 virtual void set_dw(COMP * D){ down = D;}          //  v2                    490
 virtual typ COMP_type(int l, int id){             //  v5
        comp_typ.layer=l  ;
        comp_typ.object=4 ;
        comp_typ.id_num=id;
        return comp_typ;}
public :
 OUT_base(int l, int id):COMP(),up(0),down(0){COMP_type(l,id);}
 ~OUT_base(){} //  Durring the entire simulation OUT will be linked

 virtual COMP * upstream()  {return up  ;}          //  v3                    500
 virtual COMP * downstream(){return down;}           //  v4

//  join management interface
 friend void join(COMP & UPstream, COMP & DOwnstream);};//jjoins two COMPonents
```

```
//*****************************************************************************

void join(COMP & UPstream, COMP & DOwnstream){

//   if UPstream COMPonent has not already been assigned a DOwnstream
//   COMPonent, or if it is a JUNCTION
 if((UPstream.C_type().object == 2) || !UPstream.downstream())
   UPstream.set_dw(&DOwnstream);

//   if DOwnstream COMPonent has not already been assigned an UPstream
//   COMPonent, or if it is a JUNCTION
 if((DOwnstream.C_type().object == 2) || !DOwnstream.upstream())
   DOwnstream.set_up(&UPstream);}

//*****************************************************************************

# define PARTS 9
            //  ******  Maximum number of operations any machine can do

//*********************************************************************
//** This section contains the internal structure of the components *
//** IN, LNK, JCT, WST, OUT                                        *
//*********************************************************************

class IN : public IN_base                //**** Complete
{
 private:
  PALLET * pallet    ;
  int Products[PARTS];
 public:
  Layer * L       ; //   Pointer to IN's Layer
  IN(int l, int id, int * prod):IN_base(l,id),
    pallet(0), L(0){
     for(int i=0; i<PARTS;i++){Products[i]=prod[i];}}

  ~IN(){}          //   Durring entire simulation IN will be linked

  virtual int  access(COMP *)       ; //   v6
  virtual void move(COMP *,PALLET *) ; //   v7
```

```
        int part(int k) ; //   returns 1 if part k handled by IN, else 0
        int discharge() ; //   used by LAYER_CONTROLER to move the PALLET
};                        //   at IN onto the downstream LINK. 1=success

  int IN::access(COMP * T){
            if(pallet) return 0; //   return 0 if access unavailable                    550
            else return 1;}      //   return 1 if access available

  void IN::move(COMP * T, PALLET * p){ //   assumes access is available
            pallet = p;}

      int  IN::part(int k){ return Products[k];}

      int  IN::discharge(){         //   assumes PALLET is at IN
            if(downstream()->access(this)){
              downstream()->move(this,pallet);                                        560
              pallet = 0;
              return 1;}
            else return 0;}

//*****************************************************************************

class LNK : public LNK_base              //**** Complete
{
 private:
  friend class Central_Layer;                                                         570
  friend class Output_Layer;

  PALLET ** link; //   LINK points to the first element in an array of PALLET *'s
  int lnth     ; //   length of array of PALLET *'s
  void dt() ;     //   automatic time evolution of LINK (one pallet length / second)

 public:
  Layer * L    ; //   Pointer to LNK's Layer
  LNK(int, int, int);
  ~LNK();                                                                             580
  virtual int access(COMP *)          ; //   v6
  virtual void move(COMP *, PALLET *)   ; //   v7
```

```
  float   util()          ; //  percentage of LNK occupied with pallets
  int     occu()          ; //  number of pallets in link
  int     length()        ; //  length of guidway
  PALLET * vision(int k) const ; //  allows access to the kth LNK location

  int index    ; //  Layers use index to locate, set LNK in the transition matrix
};                                                                          590

LNK::LNK(int l, int id, int dim= 1):LNK_base(l,id),lnth(dim),index(-1), L(0){
    link = (PALLET **)calloc(lnth,sizeof(PALLET *));}        //default default: length = 1

LNK::~LNK(){free(link);}

int  LNK::access(COMP * U){ //  link[0] is the first pallet * in link
              if(link[0]) return 0;  //  return 0 if access is unavailable
              else return 1;}        //  return 1 if access is available
                                                                            600
void LNK::move(COMP * U, PALLET * p){ //  assumes access is available
              link[0]= p;}

float LNK::util(){
      int count = 0;
      for(int i=0;i<lnth;i++) {if(link[i]) count +=1;}
      float temp = ((float)count)/lnth;
      return temp;}

int   LNK::occu(){                                                          610
      int count = 0;
      for(int i=0;i<lnth;i++){if(link[i]) count +=1;}
      return count;}

int   LNK::length(){return lnth;}

PALLET * LNK::vision(int k) const { return link[k];}

void  LNK::dt(){
        if(link[lnth -1]){        //  if pallet is at end of a LINK             620
          if(downstream()->access(this)){  //  access to downstream element
            downstream()->move(this,link[lnth-1]);
```

```
            for(int i=(lnth -1); i>0; i--){
                *(link + i) = *(link + i - 1);}
            *link=0;
            if(link[lnth -1]){   //   if new PALLET is at end of LINK
              link[lnth -1]->tick(this);}
            return;}
          else{   //  LINK is jammed, begins backing up
            int i=(lnth-1);                                                    630
            while(i >=0 && link[i]) i -= 1;
            if(i < 0) return; //   LINK is FULL !
            for(int j=i; j>0; j--){
                link[j] = link[j-1];}
            link[0]=0;}}
        else{
          for(int i=(lnth -1); i>0; i--){
              *(link +i) = *(link + i - 1);}
          *link=0;
          if(link[lnth -1]){   //   if new PALLET is at end of LINK              640
            link[lnth -1]->tick(this);}
          return;}}


//*****************************************************************************

class JCT : public JCT_base
{
 private:
   friend class Central_Layer;                                                 650
   friend class Output_Layer;

   void dt()          ; //   automatic time evolution of JUNCTION

 public:
   JCT(int l, int id):JCT_base(l,id){}
   ~JCT(){}            //   Durring entire simulation JCT will be linked
};

void JCT::dt(){                                                                660
```

```
    COMP * tmp;

    JCT_UP_LINK * cta = Up_List.next_sw(); //   router_store == 0 at this point
    JCT_UP_LINK * ctb = cta;
    do{
      if((cta->pallet != 0) && (tmp=cta->pallet->next_link())->access(this)){
           tmp->move(this,cta->pallet); //   load pallet with precedence
           cta->pallet = 0;             //   to its downstream link
           return;}                     //   if link is accessible                    670
      else cta=Up_List.next_sw();} while( cta != ctb);

    return;}            //   no pallet to load onto router_store

//*****************************************************************************

          //******* PARTS:  Maximum number of operations any machine can do

struct tasks{
 int part[PARTS] ;  //   lists the part types machine handles; 1= can, 0= can not   680
 int p_time[PARTS];}; //   lists the time in seconds it takes to handle each part

//*****************************************************************************

class MACHINE
{
 private:
  friend class WST;
  tasks T       ;
  int work_status ;    //   1 = BUISSY, 0 = IDLE                                    690
  int timer     ;
  float mean    ;    //   average time it takes MACHINE to process a part
  PALLET * station;

  int  access()     ; //   used by WORKSTATION to check if MACHINE is free
  void move(PALLET *); //   used by WORKSTATION to move PALLET onto MACHINE

  MACHINE(tasks t): T(t), work_status(0), timer(0), station(0){
  mean = 0;
  float a=0; float b=0;                                                            700
```

```
  for(int i= 0; i< PARTS; i++){
    if(T.part[i]){ a +=1; b +=T.p_time[i];}}
  mean = b/a;}

  ~MACHINE(){}
};

int  MACHINE::access(){
     if(station) return 0;
     else return 1;}                                                          710

void MACHINE::move(PALLET * P){ //   assumes access is available
     station    = P;
     timer      = T.p_time[station->part_num()];
     if(timer) work_status = 1;}

//*****************************************************************************

class BUFFER
{                                                                             720
 private:
   friend class WST;
   int size;        //   PALLETs BUFFER could hold
   int occu;        //   PALLETs BUFFER is holding

   PALLET ** buffer; //   array of PALLET *'s

   void insert(PALLET *);
   PALLET * remove(int)    ;
   PALLET * remove(PALLET *);                                                 730

   BUFFER(int s):occu(0), size(s){
       buffer = (PALLET **)calloc(size,sizeof(PALLET *));}
   ~BUFFER(){free(buffer);}
};

void    BUFFER::insert(PALLET * P){
        if(buffer[size-1] != 0) { cerr << "Inserting PALLET to full buffer";
        return;}
```

```
        for(int i=(size-1); i>0;i--){                                          740
          buffer[i] = buffer[i -1];}
        buffer[0] = P;
        occu += 1;
        return;}

PALLET * BUFFER::remove(int k){
        if(occu == 0){ cerr << "attempting remove PALLET from empty BUFFER";
                return 0;}
        PALLET * tmp = buffer[k];
        for(int i=k; i<(size-1);i++){                                          750
          buffer[i]=buffer[i+1];}
        buffer[size-1]=0;
        occu -= 1;
        return tmp;}


PALLET * BUFFER::remove(PALLET * P){
        int k=(size-1);
        while(buffer[k] != P){  //   First, get index of PALLET in BUFFER
          k -= 1;                                                              760
          if( k== -1){cerr << "attempting to remove PALLET not in BUFFER";
            return 0;}}
        remove(k);}

//*****************************************************************************

class WST : public WST_base
{
 private:
  friend class Central_Layer;                                                  770

  MACHINE * m         ;
  BUFFER  * b         ;

  PALLET * discipline(); //  SPD:  return PALLET to be worked on next from BUFFER
  void m_dt()         ; //   automatic time evolution of MACHINE
  void dt()           ; //   automatic time evolution of WORKSTATION
```

```
public:
  Layer * L          ; //  pointer to WST's Layer                                    780
  int  part(int k)   ; //  returns 1 if part k handled by WST MACHINE, else 0
  int  p_time(int k) ; //  returns seconds WST MACHINE takes to handle part k
  float m_time()     ; //  return mean processing time of MACHINE
  int  buff_occu()   ; //  returns # of PALLETS in WORKSTATION BUFFER
  int  buff_size()   ; //  returns # of PALLETS WST BUFFER can hold

  WST(int l, int id, int s, tasks t):L(0), WST_base(l,id){
     if(s !=0) b = new BUFFER(s) ;
     else b=0;
     m = new MACHINE(t);}                                                           790

  ~WST(){delete b;
       delete m;}

  virtual int access(COMP * U){          //vv6
            if(b){  //  WORKSTATION has a BUFFER
              if(b->buffer[(b->size)-1]){return 0;}
              else              return 1;}
           else {  //  WORKSTATION does not have a BUFFER
              if(m->station)   return 0;                                          800
              else          return 1;}}

  virtual void move(COMP * U,PALLET * P){ //  assume access is available   //v7
          if(b){  //  WORKSTATION has a BUFFER
             b->insert(P);}
          else {  //  WORKSTATION does not have a BUFFER
             m->move(P);}}

  int pole(PALLET *); //  used by LAYER_CONTROLER: returned value reflects
                 //  how long it would take the WORKSTATION to service          810
                 //  the referenced PALLET if submitted at time of polling

  int index  ; //  Layers use index to locate and set WST in transition matrix
};

int  WST::part(int k){return m->T.part[k];}
int  WST::p_time(int k){return m->T.p_time[k];}
```

```
float WST::m_time(){return m->mean;}
int   WST::buff_occu(){return b->occu;}
int   WST::buff_size(){return b->size;}                                    820

void WST::m_dt(){
    if(m->station){  //   PALLET is in MACHINE::station
      if(m->work_status){      //   PALLET is being processed
        (m->timer) -= 1;
        if((m->timer) <= 0 ) {m->timer = 0; m->work_status = 0;}
        return;}
      else{        //   PALLET is in MACHINE::station, proccessing is complete
        if(downstream()->access(this)){ //   check access to downstream element
          downstream()->move(this,m->station);                             830
          m->station = 0;
          return;}    //   downstream free, move successful
        else return;}} //   downstream blocked, move unsuccessful
    else return;}       //   no PALLET is in MACHINE::station

void WST::dt(){
     if(b && (m->access())){//*** WORKSTATION has a BUFFER && MACHINE is accessible
      if(b->occu){
      PALLET * tmp = discipline(); //*** discipline should also delete PALLET *
      if(tmp) m->move(tmp);}}                                              840
     m_dt();}

//*****************************************************************************

class OUT_ORDER
{
 private:
  int order_id;                 //   id number of ORDER at OUT_ORDER
  TCK due_date;                   //   due date of ORDER at OUT_ORDER
                                                                           850
  const int order_sz;             //   size of ORDER
      int completed_sz;           //   size of completed ORDER at OUT_ORDER

  const COMPOSITION order_comp;     //   composition of ORDER
      COMPOSITION completed_comp; //   composition of completed ORDER at OUT_ORDER
```

```
 OUT_ORDER(const OUT_ORDER &);          //  copy constructor, not implemented
 OUT_ORDER & operator=(const OUT_ORDER &); //  assignment, not implemented

 friend class OUT_QUEUE;                                                    860

 OUT_ORDER * next;              //  * to fascilitate singly linked OUT_QUEUE
 OUT_ORDER(const IN_ORDER &, OUT_ORDER *);
 ~OUT_ORDER(){}

public: //  Interface Functions
 int id() const    :           //  return OUT_ORDER::order_id
 TCK ck_due() const:           //  return OUT_ORDER::due_date

 int o_size() const;           //  return OUT_ORDER::order_sz                 870
 int c_size() const;           //  return OUT_ORDER::completed_sz

 int o_comp(int PART) const;   //  return OUT_ORDER::order_comp
 int c_comp(int PART) const;   //  return OUT_ORDER::completed_comp

 int change_comp(int PART,int CHANGE); //  change the number of part PART in
};                             //  OUT_ORDER by change

OUT_ORDER::OUT_ORDER(const IN_ORDER & in_order, OUT_ORDER * n):completed_sz(0),
   completed_comp(), order_sz(in_order.o_size()), order_comp(in_order.original_c()),   880
   next(n){
        order_id   = in_order.id();
        due_date   = in_order.ck_due();}


int OUT_ORDER::id() const {return order_id;}
TCK OUT_ORDER::ck_due() const {return due_date;}
int OUT_ORDER::o_size() const {return order_sz;}
int OUT_ORDER::c_size() const {return completed_sz;}
int OUT_ORDER::o_comp(int PART) const {return order_comp.comp(PART);}          890
int OUT_ORDER::c_comp(int PART) const {return completed_comp.comp(PART);}

int OUT_ORDER::change_comp(int PART, int CHANGE){
        completed_comp.comp(PART,CHANGE);
        completed_sz += CHANGE;
```

```
        if(completed_sz==order_sz) return 1; //   1 indicates ORDER is complete
        return 0;}

//*****************************************************************************
                                                                          900
class OUT_QUEUE
{
 private:
  OUT_ORDER * c   ; //   iterator currency
  OUT_ORDER * lead; //   first object in out_Q

  OUT_QUEUE(const OUT_QUEUE &);          //   copy constructor, not implemented
  OUT_QUEUE & operator=(const OUT_QUEUE &); //   assignment, not implemented

 public:                                                                  910
  OUT_QUEUE();
  ~OUT_QUEUE();

  OUT_ORDER * insert(const IN_ORDER &);//   put  ORDER in QUEUE, return OUT_ORDER *
  int       remove(int order_id)   ;//   rm OUT_ORDER from QUEUE, return order_id
  void       reset()               ;//   reset currency
  OUT_ORDER * isIn(int order_id)     ;//   returns * to order_id OUT_ORDER, else 0
  OUT_ORDER * next()                ;//   returns * to OUT_ORDER currently referenced
};                              //   by c then increments c, or returns 0 if end
                                //   of QUE is reached                     920

OUT_QUEUE::OUT_QUEUE():c(0),lead(0) {}

OUT_QUEUE::~OUT_QUEUE(){
      if(lead) cout << "Terminating OUT_QUEUE not empty\n";
      while(lead !=0){
        OUT_ORDER *t=lead->next;
        delete lead;
        lead = t;}}
                                                                          930

OUT_ORDER *  OUT_QUEUE::insert(const IN_ORDER & I){
      lead = new OUT_ORDER(I,lead);
      return lead;}
```

```
int OUT_QUEUE::remove(int order_id){
      OUT_ORDER * curr = lead;
      OUT_ORDER * prev = 0;
      if(!lead) cout <<"No orders to remove, OUT_QUEUE is empty.\n";
      while(curr){                                                          940
        if(curr->order_id == order_id){
          if(prev){
            prev->next=curr->next;}
          else{
            lead=curr->next;}
          if(c==curr) c=curr->next;
          delete curr;
          return order_id;}
        prev= curr;
        curr = curr->next;}                                                 950
      cout <<"OUT_ORDER was not in OUT_QUEUE\n";
      return 0;}

void OUT_QUEUE::reset(){ c= lead;}

OUT_ORDER * OUT_QUEUE::isIn(int order_id){
      OUT_ORDER * q;
      reset();
      while((q=next()) !=0){
        if(q->order_id == order_id) return q;                               960
      }
      cout << "ORDER #"<<order_id<< " is not in OUT_QUEUE\n";
      return 0;}

OUT_ORDER * OUT_QUEUE::next(){
      if(c){
        OUT_ORDER * r= c;
        c=c->next;
        return r;}
      else return 0;}                                                       970

//*****************************************************************************
```

```
class OUT : public OUT_base
{
 private:
  friend class Output_Layer;
  friend int main();
  OUT_QUEUE Q  ;    //  stores a list of OUT_ORDERS
  int o_num    ;    //  total number of ORDERS dedicated to OUT                    980
  int p_num    :    //  total number of pallets dedicated to OUT

  OUT_ORDER * insert(const IN_ORDER &); //  New ORDER assigned to OUT
  int remove(int order_id) ; //  performs operations necesary in deleting an order
                        //  removes order from out_Q, ORDER_Q
 public:
Layer * L      ;    //  pointer to OUT's Layer
virtual int access(COMP *):       //  access to OUT is always granted       //v6
virtual void move(COMP *,PALLET *);//  here pallet is not stored, but deleted  //v7
                                                                               990
  OUT(int l, int id):OUT_base(l,id),Q(), o_num(0), p_num(0), L(0){}
  ~OUT(){}

  int pole(const IN_ORDER *);    //  SPD:  used by LAYER_CONTROLER
  OUT_ORDER * inf(int);          //  used by LAYER_CONTROLER to check the status
                          //  of an ORDER identified by its order_id

  int index  ; //  Layers use index to locate and set OUT in transition matrix
};
                                                                               1000
int OUT::access(COMP * U){return 1;} //  access always granted

void OUT::move(COMP * U, PALLET * P){
      Q.reset(); //  First, search for OUT_ORDER PALLET belongs to
      OUT_ORDER * tmp;
      while( (tmp = Q.next()) != 0){  //  Register arival of PALLET,
      if(tmp->id() == P->order_num()){ //  if ORDER complete, use done(id)
        if(tmp->change_comp(P->part_num(),1)) L->done(this, tmp->id());
        p_num -=1; P->can_delete();
        delete P; //  Remove Pallet from system and Reclaim Memory              1010
        return;}}
      cerr<<"PALLET has reached incorrect OUT";
```

```
        return;}

OUT_ORDER * OUT::insert(const IN_ORDER & order){
        OUT_ORDER * tmp = Q.insert(order);
        o_num += 1;
        p_num += (tmp->o_size());
        return tmp;}                                                              1020

int OUT::remove(int order_id){  //   LAYER CONTROLER removes a completed ORDER
        if(Q.remove(order_id)==0) return 0; //   ORDER was not in OUT_QUEUE
        o_num -= 1;
        return order_id;}

OUT_ORDER * OUT::inf(int order_id){
        OUT_ORDER *tmp = Q.isIn(order_id);
        return tmp;}

//*************************************************************************************** 1030
//***************************************************************************************
//******                                                    ******
//******              LAYERS_CONTROL AND COMUNICATION               ******
//******                                                    ******
//***************************************************************************************
//***************************************************************************************

struct Next_up{int part; IN_ORDER * order;};

class IN_table_link                                                               1040
{
 private:
   friend class IN_table ;
   IN_table_link * next;
   IN          * dat ;
   IN_table_link(IN * d, IN_table_link * n):dat(d),next(n){}
   ~IN_table_link(){}
 public:
   IN * data(){return dat;}};
                                                                                  1050
//***************************************************************************************
```

```
class Input_Layer;

class IN_table            //   Singly linked list
{
 private:
  IN_table_link * lead ;  //   points to head of list
  IN_table_link * c    ;  //   iterator currency
                                                                    1060
  friend class Input_Layer;
  IN * insert(IN *);
  int s;

 public:
  int  size()      ;
  IN * next()      ;
  IN * isIn(typ)   ;
  IN * isIn(COMP *);
  void reset()     ;                                              1070

  IN_table():lead(0), c(0), s(0){}
  ~IN_table(){
     while(lead !=0){
       IN_table_link *t = lead->next;
       delete lead->dat;
       delete lead;
       lead = t;}}};

int  IN_table::size(){return s;}                                    1080

IN * IN_table::insert(IN * in){
        lead = new IN_table_link(in,lead);
        s += 1;
        return lead->dat;}


IN * IN_table::next(){
        if(c){
          IN_table_link *t=c;                                       1090
```

```
        c=c->next;
        return t->dat;}
      else return 0;}

IN * IN_table::isIn(typ T){
      IN * q;
      reset();
      while((q=next()) != 0){
        typ S = q->C_type();
        if( S.id_num==T.id_num && S.object==T.object && S.layer==T.layer){
          return q;}
          }
      return 0;}

IN * IN_table::isIn(COMP *cmp){
      IN * q;  IN * test = (IN *) cmp;
      reset();
      while((q=next()) != 0){
       if(q==test) return q;
         }
      return 0;}

void IN_table::reset(){ c=lead;}

//*****************************************************************************

class Central_Layer;
class Output_Layer ;

class LNK_table_link
{
 private:
  friend class LNK_table ;
  LNK_table_link * next;
  LNK          * dat ;
  LNK_table_link(LNK * d, LNK_table_link * n):dat(d),next(n){}
  ~LNK_table_link(){}
 public:
  LNK * data(){return dat;}};
```

```
                                                                                    1130
//*****************************************************************************

class LNK_table            //*** Singly linked list
{
 private:
  LNK_table_link * lead ;  //*** points to head of list
  LNK_table_link * c    ;  //*** iterator currency

  friend class Central_Layer;
  friend class Output_Layer ;                                                       1140
  friend class TRAN         ;
  LNK * insert(LNK *);
  int s;                   //*** number of elements in LNK_table

 public:
  int   size()      ;
  LNK * next()      ;
  LNK * isIn(typ)   ;
  LNK * isIn(COMP *);
  void  reset()     ;                                                               1150
  void  clear()     ;      //*** removes all elements from LNK_table
  void  remove()    ;      //*** removes lead element LNK in LNK_table

  LNK_table():lead(0), c(0), s(0){}
  ~LNK_table(){
     while(lead !=0){
       LNK_table_link *t = lead->next;
       delete lead->dat;
       delete lead;
       lead = t;}}};                                                                1160

 int   LNK_table::size(){return s;}

 LNK * LNK_table::insert(LNK * lnk){
         lead = new LNK_table_link(lnk,lead);
         s +=1;
         return lead->dat;}
```

```
LNK * LNK_table::next(){
        if(c){                                                          1170
          LNK_table_link *t=c;
          c=c->next;
          return t->dat;}
        else return 0;}

LNK * LNK_table::isIn(typ T){
        LNK * q;
        reset();
        while((q=next()) != 0){
          typ S = q->C_type();                                          1180
          if( S.id_num==T.id_num && S.object==T.object && S.layer==T.layer){
            return q;}
            }
        return 0;}

LNK * LNK_table::isIn(COMP *cmp){
        LNK * q;  LNK * test = (LNK *) cmp;
        reset();
        while((q=next()) != 0){
          if(q==test) return q;                                         1190
            }
        return 0;}

void LNK_table::reset(){ c=lead;}

void LNK_table::clear(){
        s=0;
        c=0;
        while(lead !=0){
          LNK_table_link *t= lead->next;                                1200
          delete lead;              //*** note:  I am not deleting the dat (LINK)
          lead =t    ;}}            //***        only the LNK_table_link

void LNK_table::remove(){
        if(lead !=0){
          LNK_table_link *t= lead->next;
          delete lead;
```

```
        lead = t;
        s -=1;}
      else cout <<"attempting to remove LNK_table_link from empty LNK_table\n";}          1210

//*****************************************************************************

class JCT_table_link
{
 private:
  friend class JCT_table ;
  JCT_table_link * next;
  JCT            * dat ;
  JCT_table_link(JCT * d, JCT_table_link * n):dat(d),next(n){}                          1220
  ~JCT_table_link(){}
 public:
  JCT * data(){return dat;}};

//*****************************************************************************

class JCT_table            //   Singly linked list
{
 private:
  JCT_table_link * lead ;  //   points to head of list                                1230
  JCT_table_link * c    ;  //   iterator currency

  friend class Central_Layer;
  friend class Output_Layer;
  JCT * insert(JCT *);
  int s;

 public:
  int   size()     ;
  JCT * next()     ;                                                                    1240
  JCT * isIn(typ)  ;
  JCT * isIn(COMP *);
  void reset()     ;

  JCT_table():lead(0), c(0), s(0){}
  ~JCT_table(){
```

```
    while(lead !=0){
     JCT_table_link *t = lead->next;
     delete lead->dat;
     delete lead;                                                    1250
     lead = t;}}};

int   JCT_table::size(){ return s;}

JCT * JCT_table::insert(JCT * jct){
       lead = new JCT_table_link(jct,lead);
       s +=1;
       return lead->dat;}

JCT * JCT_table::next(){                                           1260
       if(c){
         JCT_table_link *t=c;
         c=c->next;
         return t->dat;}
       else return 0;}

JCT * JCT_table::isIn(typ T){
       JCT * q;
       reset();
       while((q=next()) != 0){                                     1270
         typ S = q->C_type();
         if( S.id_num==T.id_num && S.object==T.object && S.layer==T.layer){
           return q;}
           }
       return 0;}

JCT * JCT_table::isIn(COMP *cmp){
       JCT* q;  JCT * test = (JCT *) cmp;
       reset();
       while((q=next()) != 0){                                     1280
         if(q==test) return q;
           }
       return 0;}

void JCT_table::reset(){ c=lead;}
```

```
//*****************************************************************************

class WST_table_link
{                                                                          1290
 private:
  friend class WST_table ;
  WST_table_link * next;
  WST            * dat ;
  WST_table_link(WST * d, WST_table_link * n):dat(d),next(n){}
  ~WST_table_link(){}
 public:
  WST * data(){return dat;}};

//***************************************************************************** 1300

class WST_table            //   Singly linked list
{
 private:
  WST_table_link * lead ;  //   points to head of list
  WST_table_link * c    ;  //   iterator currency

  friend class Central_Layer;
  friend class Network     ;
  WST * insert(WST *);                                                     1310
  int s;

 public:
  int   size()     ;
  WST * next()     ;
  WST * isIn(typ)  ;
  WST * isIn(COMP *);
  void clear()     ;
  void reset()     ;
                                                                           1320
  WST_table():lead(0), c(0), s(0){}
  ~WST_table(){
     while(lead !=0){
       WST_table_link *t = lead->next;
```

```
        delete lead->dat;
        delete lead;
        lead = t;}}};

int   WST_table::size(){return s;}

                                                                        1330
WST * WST_table::insert(WST * wst){
        lead = new WST_table_link(wst,lead);
        s +=1;
        return lead->dat;}

WST * WST_table::next(){
        if(c){
          WST_table_link *t=c;
          c=c->next;
          return t->dat;}                                               1340
        else return 0;}

WST * WST_table::isIn(typ T){
        WST * q;
        reset();
        while((q=next()) != 0){
          typ S = q->C_type();
          if( S.id_num==T.id_num && S.object==T.object && S.layer==T.layer){
            return q;}
            }                                                           1350
        return 0;}

WST * WST_table::isIn(COMP *cmp){
        WST * q;  WST * test = (WST *) cmp;
        reset();
        while((q=next()) != 0){
          if(q==test) return q;
            }
        return 0;}
                                                                        1360
void WST_table::clear(){
    s=0;
    c=0;
```

```
    while(lead !=0){
      WST_table_link *t = lead->next;
      delete lead;
      lead = t;}}

void WST_table::reset(){ c=lead;}
                                                                                    1370
//*************************************************************************************

class OUT_table_link
{
 private:
  friend class OUT_table ;
  OUT_table_link * next;
  OUT           * dat ;
  OUT_table_link(OUT * d, OUT_table_link * n):dat(d),next(n){}
  ~OUT_table_link(){}                                                               1380
 public:
  OUT * data(){return dat;}};

//*************************************************************************************

class OUT_table            //   Singly linked list
{
 private:
  OUT_table_link * lead ;  //   points to head of list
  OUT_table_link * c    ;  //   iterator currency                                  1390

  friend class Output_Layer;
  OUT * insert(OUT *);
  int s;

 public:
  int   size()     ;
  OUT * next()     ;
  OUT * isIn(typ)  ;
  OUT * isIn(COMP *);                                                               1400
  void reset()     ;
```

```
 OUT_table():lead(0), c(0), s(0){}
 ~OUT_table(){
    while(lead !=0){
     OUT_table_link *t = lead->next;
     delete lead->dat;
     delete lead;
     lead = t;}}};

int   OUT_table::size(){return s;}

OUT * OUT_table::insert(OUT * out){
       lead = new OUT_table_link(out,lead);
       s += 1;
       return lead->dat;}

OUT * OUT_table::next(){
       if(c){
        OUT_table_link *t=c;
        c=c->next;
        return t->dat;}
       else return 0;}

OUT * OUT_table::isIn(typ T){
       OUT * q;
       reset();
       while((q=next()) != 0){
        typ S = q->C_type();
        if( S.id_num==T.id_num && S.object==T.object && S.layer==T.layer){
          return q;}
         }
       return 0;}

OUT * OUT_table::isIn(COMP *cmp){
      OUT * q;  OUT * test = (OUT *) cmp;
      reset();
      while((q=next()) != 0){
       if(q==test) return q;
         }
      return 0;}
```

```
void OUT_table::reset(){ c=lead;}

//*****************************************************************************

class PATH
{
 public:
  int size;                                                          1450
  LNK **list;

  PATH():size(0), list(0){}     ; //   default constructor
  PATH(LNK_table &)             ; //   constructor with list argument
  PATH(const PATH &)            ; //   Copy constructor
  PATH & operator=(const PATH &); //   assignment
  ~PATH()                       ; //   destructor
};

 PATH::PATH(LNK_table & lt){                                         1460
     size = lt.size();
     lt.reset();
     list = (LNK **)calloc(size,sizeof(LNK *));
     for(int i= size-1; i>=0 ; i--) list[i] = lt.next();}

 PATH::PATH(const PATH & source){
     size = source.size;
     list = (LNK **)calloc(size,sizeof(LNK *));
     for(int i=0; i<size; i++) list[i] = source.list[i];}
                                                                     1470
 PATH & PATH::operator=(const PATH & source){
     if(this != &source){
       size = source.size;
       free(list);
       list = (LNK **)calloc(size,sizeof(LNK *));
       for(int i=0; i<size; i++) list[i] = source.list[i];}
     return *this;}

 PATH::~PATH(){free(list);}
                                                                     1480
```

```
//*****************************************************************************

class W_PATH
{
 public:
  int size;
  WST **list;

  W_PATH():size(0), list(0){}       ; //   default constructor
  W_PATH(WST_table &)               ; //   constructor with list argument
  W_PATH(const W_PATH &)            ; //   Copy constructor
  W_PATH & operator=(const W_PATH &); //   assignment
  ~W_PATH()                         ; //   destructor
};

 W_PATH::W_PATH(WST_table & wt){
     size = wt.size();
     wt.reset();
     list = (WST **)calloc(size,sizeof(WST *));
     for(int i= size-1; i>=0 ; i--) list[i] = wt.next();}

 W_PATH::W_PATH(const W_PATH & source){
     size = source.size;
     list = (WST **)calloc(size,sizeof(WST *));
     for(int i=0; i<size; i++) list[i] = source.list[i];}

 W_PATH & W_PATH::operator=(const W_PATH & source){
     if(this != &source){
       size = source.size;
       free(list);
       list = (WST **)calloc(size,sizeof(WST *));
       for(int i=0; i<size; i++) list[i] = source.list[i];}
     return *this;}

 W_PATH::~W_PATH(){free(list);}

//*****************************************************************************

struct Next{int part; IN_ORDER * in_order;};
```

```
                                                                          1520
class Input_Layer : public Layer
{
 private:
  friend int main() ;
  int layer_id    ;
  IN_table ins    ;
  IN * add_in(IN * in){ in->L= this; return ins.insert(in);}

  void    dt();                  //  automatic time evolution of Input_Layer
  PALLET * make_pallet(Next);    //  Creates a "raw" PALLET using Next          1530
  Next discipline(IN *,IN_QUEUE &); //  Method of Choosing PALLET to load at IN
                                 //  if no Next PALLET, Next={0,0}    *User Spec
  int     discharge(IN *);       //  Condition must satisfy to discharge PALLET
                                 //  1= discharge, 0= do not discharge *User Spec
 public:
  Input_Layer(int i):layer_id(i){}          //  Constructor
  ~Input_Layer(){}                          //  Destructor
  int id(){return layer_id;}
  IN_table * in_table(){ return &ins;}};
                                                                          1540
void  Input_Layer::dt(){
      ins.reset();
      while(IN * q = ins.next()){           //  look at new IN
        if(q->access(q)){                   //  is IN free ?
          Next n = discipline(q,Q);         //  yes, IN is free
          if(n.in_order){                   //  can we load in with a PALLET ?
            q->move(q,make_pallet(n));}}    //  yes, then load it
          else{                             //  no, IN is not free
            if(q->downstream()->access(q) && discharge(q))
              q->discharge();}}}                                            1550

PALLET * Input_Layer::make_pallet(Next n){
      IN_ORDER * q = n.in_order;
      PALLET * p = new PALLET(q->id(), n.part, q->ck_due(), q->out_stream());
      n.in_order->change_comp(n.part,-1); //  PALLET in system logged in IN_ORDER
       return p;}

//*****************************************************************************
```

```
class Central_Layer: public Layer                                          1560
{
 private:
  friend int main();
  friend class Network;

  int layer_id     ;  //   layer identification number
  int input_dim    ;  //   number of LNKs with UPstream COMPonents in another layer
  int output_dim   ;  //   number of WORKSTATIONS in Central_Layer
  int networked    ;  //   1 => route based scheduler, 0=> local feedback scheduler
  Basenet *N       ;  //   pointer to Central_Layer's network                1570
  LNK_table lnks   ;
  JCT_table jcts   ;
  WST_table wsts   ;

  LNK * add_lnk(LNK * lnk){lnk->L = this; return lnks.insert(lnk);}
  JCT * add_jct(JCT * jct){jct->L = this; return jcts.insert(jct);}
  WST * add_wst(WST * wst){wst->L = this; return wsts.insert(wst);}

  void  dt();          //   automatic time evolution for Central_Layer
                                                                           1580
  LNK_table TMP    ;                 //   Used to make transit matrix
  void popper(COMP *, COMP *, LNK *); //   Used to make transit matrix
  void set(LNK *, WST *)             ; //   Used to make transit matrix
  void make()                      ; //   Used to make transit matrix

 public:

  float   cost(PATH *, PALLET *, WST*); //   SPD:  cost of routing decision
  PATH ** tran     ;
                                                                           1590
  Central_Layer(int i):layer_id(i), input_dim(0), output_dim(0), tran(0),
                networked(0), N(0){}
  ~Central_Layer(){
    for(int i =0; i< output_dim; i++) delete [] tran[i];
    free(tran); cout <<"Central_Layer Transition Matrix Deleted\n";}

  int id(){return layer_id;}
```

```
 int in_dim() {return input_dim;}
 int out_dim(){return output_dim;}
 LNK_table * lnk_table(){ return &lnks;}                                    1600
 JCT_table * jct_table(){ return &jcts;}
 WST_table * wst_table(){ return &wsts;}

 void   make_transit()              ;
 PATH * transit(LNK *, WST *)       ; //   PATH from LNK to WST, 0 if nonexistant

 virtual void  inform(JCT *, PALLET *) ; //   Layer routes PALLET from JCT
 WST    * decentral_1(JCT *, PALLET*)  ;

 void   fix(PALLET *, PATH *, WST *)   ; //   Layer fixes PALLET PATH and WST   1610
 void   fix(PALLET *, PATH *, W_PATH *); //    ||    ||    ||    ||  and W_PATH
 void   fix(PALLET *, PATH *)          ; //    ||    ||    ||    ||
};

void Central_Layer::dt(){
    lnks.reset();
    while(LNK * q = lnks.next()) q->dt();
    jcts.reset();
    while(JCT * q = jcts.next()) q->dt();
    wsts.reset();                                                           1620
    while(WST * q = wsts.next()) q->dt();}

void Central_Layer::make(){
    LNK * q       ;
    lnks.reset()  ;
    while(q= lnks.next()){
     TMP.clear();
     if(q->upstream() && (q->upstream()->C_type().layer != q->C_type().layer)){
       TMP.insert(q);
       if(q->downstream()->C_type().object == 3){  //   q upstream of WST      1630
         set(q,(WST *) q->downstream());}
       else popper(q->downstream(),0,q);}          //   q upstream of JCT
    }
   }

void Central_Layer::popper(COMP * curr_jct, COMP * prev_jct, LNK *q){
```

```
    LNK * lnk = (LNK *) curr_jct->downstream();
    LNK * b   = lnk;
    do{
      TMP.insert(lnk);                                                    1640
      if(lnk->downstream()->C_type().object == 3){  //   lnk upstream of WST
        set(q,(WST *) lnk->downstream());
        TMP.remove();}
      if(lnk->downstream()->C_type().object == 2){  //   lnk upstream of JCT
        if(lnk->downstream()  == prev_jct) TMP.remove();
        else popper(lnk->downstream(),curr_jct,q);}
      lnk = (LNK *) curr_jct->downstream();} while(lnk != b);
    TMP.remove();}

void Central_Layer::set(LNK * lnk, WST * wst){                            1650
    int i =  wst->index;
    int j =  lnk->index;
    PATH  tmp(TMP);
    tran[i][j]=tmp;}

void Central_Layer::make_transit(){
    lnks.reset(); LNK * q;
    int j=0;
    while(q = lnks.next()){
      if( q->upstream() &&                                                1660
        (q->upstream()->C_type().layer != q->C_type().layer)){
          input_dim +=1; q->index   =j; j +=1;}}

    wsts.reset(); WST * w;
    j=0;
    while(w = wsts.next()){
          output_dim +=1; w->index =j; j +=1;}

    tran = (PATH **) calloc(output_dim,sizeof(PATH *));   //   rows: WSTs
    for(int i=0; i< output_dim; i++){                                     1670
      tran[i]= new PATH[input_dim];}                  //   colums: LNKs

    make();}

PATH * Central_Layer::transit(LNK * lnk, WST * wst){
```

```
    int i = wst->index;
    int j = lnk->index;
    return &tran[i][j];}

void Central_Layer::inform(JCT * jct, PALLET * pall){                    1680
  if(pall->wst_need()){
    LNK * b = pall->last_tick_on();
    if(networked){
      W_PATH wpa = N->centralized_1(this,jct,pall); //   WST *'s placed in W_PATH
      fix(pall,transit(b,wpa.list[0]),&wpa);}
    else{
      WST * NEXT_WST;
      NEXT_WST = decentral_1(jct,pall);
      fix(pall,transit(b,NEXT_WST), NEXT_WST);
      return;}                                                           1690
       }
  else{
    PATH * p = transit(pall->last_tick_on(), pall->next_wst());
    fix(pall, p);
    return;}}

WST * Central_Layer::decentral_1(JCT * jct, PALLET * pall){
    float NEXT; WST * NEXT_WST; WST * q; PATH * p;
    int a = pall->part_num(); LNK * b = pall->last_tick_on();
                                                                         1700
    wsts.reset();
    do{ //*** Find First Acceptable WST
      q = wsts.next(); if(q==0) cerr<<"PALLET has reached dead end\n";
      p = transit(b,q);} while( !(q->part(a) == 1 && p->size != 0));

      NEXT = cost(p,pall,q);
      NEXT_WST = q;

    while( q = wsts.next()){ //*** Find Best WST
      p = transit(b,q);                                                  1710
      if(q->part(a) == 1 && p->size != 0){
        float n = cost(p,pall,q);
        if(n < NEXT){
          NEXT = n; NEXT_WST = q;}}}
```

```
        return NEXT_WST;}

void Central_Layer::fix(PALLET * pall, PATH * pa, WST * next_wst){
          if(next_wst==0) cerr <<"WORKSTATION was not found for PALLET";
          pall->fix_direc(1,&next_wst);
          pall->fix_direc(pa->size,pa->list);}                                    1720

void Central_Layer::fix(PALLET * pall, PATH * pa, W_PATH * wpa){
          pall->fix_direc(wpa->size,wpa->list);
          pall->fix_direc(pa->size,pa->list);}

void Central_Layer::fix(PALLET * pall, PATH * pa){
          pall->fix_direc(pa->size,pa->list);}

//*****************************************************************************
                                                                                  1730
class Output_Layer : public Layer
{
 private:
  friend int main()                    ;

  int layer_id     ;  //   layer identification number
  int input_dim    ;  //   number of LNKs with UPstream COMPonents in another layer
  int output_dim   ;  //   number of WORKSTATIONS in Output_Layer
  LNK_table lnks   ;
  JCT_table jcts   ;                                                              1740
  OUT_table outs   ;

  LNK * add_lnk(LNK * lnk){lnk->L = this; return lnks.insert(lnk);}
  JCT * add_jct(JCT * jct){jct->L = this; return jcts.insert(jct);}
  OUT * add_out(OUT * out){out->L = this; return outs.insert(out);}

  void  dt();          //    automatic time evolution for Output_Layer

  LNK_table TMP    ;              //   Used to make transit matrix
  void popper(COMP *, COMP *, LNK *); //   Used to make transit matrix              1750
  void set(LNK *, OUT *)          ; //   Used to make transit matrix
  void make()                   ; //   Used to make transit matrix
```

```
public:
 PATH ** tran    ;
 FILE * FP       ;
 char * name     ,

 Output_Layer(int i,char * n):layer_id(i), input_dim(0),
    output_dim(0), tran(0), name(n){                                          1760
    if((FP = fopen(name,"w")) == NULL) cerr<<"cant open data file"<< name <<"\n";}

 ~Output_Layer(){
  for(int i =0; i< output_dim; i++) delete [] tran[i];
  free(tran);
  if(fclose(FP) != 0) cerr <<"error closing data file"<< name <<"\n";
  cout<<"Output_Layer Transition Matrix Deleted\n";}

 int id(){return layer_id;}
 int in_dim() {return input_dim;}                                             1770
 int out_dim(){return output_dim;}
 LNK_table * lnk_table(){ return &lnks;}
 JCT_table * jct_table(){ return &jcts;}
 OUT_table * out_table(){ return &outs;}

 void make_transit()           ;
 PATH * transit(LNK *, OUT *); //   map from LNK to OUT if it exists, else 0

 OUT  * destination(IN_ORDER *); //   SPD:  used by main to assign ORDER an OUTport
                                                                              1780
 virtual void  inform(JCT *, PALLET *) ;//   Layer routes PALLET from JCT
 virtual void  done(OUT *, int)      ;//   used to inform Layer order is done
 void   fix(PALLET *, PATH *)      ; //   Layer fixes PALLET PATH
};

void Output_Layer::dt(){
    lnks.reset();
    while(LNK * q = lnks.next()) q->dt();
    jcts.reset();
    while(JCT * q = jcts.next()) q->dt();}                                    1790

void Output_Layer::make(){
```

```
    LNK * q       ;
    lnks.reset()  ;
    while(q= lnks.next()){
     TMP.clear();
     if(q->upstream() && (q->upstream()->C_type().layer != q->C_type().layer)){
      TMP.insert(q);
      if(q->downstream()->C_type().object == 4){  //   q upstream of OUT
       set(q,(OUT *) q->downstream());}                                          1800
      else popper(q->downstream(),0,q);}          //   q upstream of JCT
    }
   }

void Output_Layer::popper(COMP * curr_jct, COMP * prev_jct, LNK *q){
     LNK * lnk = (LNK *) curr_jct->downstream();
     LNK * b   = lnk;
     do{
      TMP.insert(lnk);
      if(lnk->downstream()->C_type().object == 4){  //   lnk upstream of OUT     1810
       set(q,(OUT *) lnk->downstream());
       TMP.remove();}
      if(lnk->downstream()->C_type().object == 2){  //   lnk upstream of JCT
       if(lnk->downstream()  == prev_jct) TMP.remove();
       else popper(lnk->downstream(),curr_jct,q);}
      lnk = (LNK *) curr_jct->downstream();} while(lnk != b);
     TMP.remove();}

void Output_Layer::set(LNK * lnk, OUT * out){
    int i =  out->index;                                                         1820
    int j =  lnk->index;
    PATH tmp(TMP);
    tran[i][j]= tmp;}

void Output_Layer::make_transit(){
     lnks.reset(); LNK * q;
     int j=0;
     while(q = lnks.next()){
      if( q->upstream() &&
        (q->upstream()->C_type().layer != q->C_type().layer)){                   1830
          input_dim +=1; q->index = j; j += 1;}}
```

```
    outs.reset(); OUT * o;
    j=0;
    while(o = outs.next()){
        output_dim +=1; o->index =j; j += 1;}

    tran = (PATH **)calloc(output_dim,sizeof(PATH *)); //   rows: OUTs
    for(int i=0; i< output_dim; i++){
      tran[i]= new PATH[input_dim];}                  //   colums: LNKs
                                                                          1840
    make();}

PATH * Output_Layer::transit(LNK * lnk, OUT * out){
    int i =  out->index;
    int j =  lnk->index;
    return &tran[i][j];}

void Output_Layer::inform(JCT * jct, PALLET * pall){
      PATH * p = transit((LNK *) jct->upstream(), pall->out_ptr());
      fix(pall,p);                                                        1850
      return;}

void Output_Layer::fix(PALLET * pall, PATH * pa){
        pall->fix_direc(pa->size,pa->list);}

//   When Order Is Done

int stopwatch(TCK start,TCK stop){ //   seconds in stop - start
    int tmp1 = 24*3600*stop.day +
               3600*stop.hour +                                           1860
                 60*stop.minute +
                   stop.second ;

    int tmp2 = 24*3600*start.day +
               3600*start.hour +
                 60*start.minute +
                   start.second ;

    return tmp1-tmp2;}
                                                                          1870
```

```
int stopwatch(TCK tock){ //   seconds in tock
    int tmp1 = 24*3600*tock.day +
               3600*tock.hour +
                 60*tock.minute +
                   tock.second ;
    return tmp1;}


void Output_Layer::done(OUT * out, int id){
      //   stuff which will be used for policy comparison                    1880
      //   functions should go here

      out->remove(id); //   delete OUT_QUEUE
      TCK stop  = t.ck();
      TCK start = Q.isIn(id)->ck_issue();
      int size  = Q.isIn(id)->o_size();

      cout<<"Order #"<<id<<" completed at Time:     "
        <<stop.day<<"/"<<stop.hour<<":"<<stop.minute
        <<":"<<stop.second<<'\n';                                            1890

      int T2 = stopwatch(start.stop);
      int T1 = stopwatch(start);

      //       (start):(stop-start):(size)
      fprintf(FP,"%d %d %d\n",T1,T2,size);

      Q.remove(id);   //   delete IN_QUEUE
      return;}
                                                                             1900
//*****************************************************************************


class net_table_link
{
 private:
   friend class net_table ;
   net_table_link   * next;
   Central_Layer    * dat ;
```

```
  net_table_link(Central_Layer * d, net_table_link * n):dat(d),next(n){}
  ~net_table_link(){}
 public:
  Central_Layer * data(){return dat;}};

//*****************************************************************************

class net_table          //  Singly linked list
{
 private:
  net_table_link * lead ; //  points to head of list
  net_table_link * c    ; //  iterator currency

  friend class Network;
  Central_Layer * insert(Central_Layer *);
  int s;

 public:
  int   size()          ;
  Central_Layer * next()    ;
  Central_Layer * isIn(int) ;
  void reset()          ;

  net_table():lead(0), c(0), s(0){}
  ~net_table(){
     while(lead !=0){
       net_table_link *t = lead->next;
       //*** there was a line here deleting all included Central_Layers
       delete lead;
       lead = t;}}};

 int   net_table::size(){return s;}

 Central_Layer * net_table::insert(Central_Layer * ct){
  lead = new net_table_link(ct,lead);
  s +=1;
  return lead->dat;}
```

```
Central_Layer * net_table::next(){
        if(c){                                                          1950
          net_table_link *t=c;
          c=c->next;
          return t->dat;}
        else return 0;}

Central_Layer * net_table::isIn(int id){
        Central_Layer * q;
        reset();
        while((q=next()) != 0){
          if(q->id() == id) return q;}                                  1960
        return 0;}

void net_table::reset(){ c=lead;}

//*****************************************************************************

class Network : public Basenet
{
private:
 net_table CLL ; //   table of central Layers                          1970
 WST_table WL  ; //   table of workstations for Central_Layer
 int networked ; //   1=> route based scheduler, 0=> local feedback scheduler

 virtual W_PATH centralized_1(Central_Layer *cent_lay, JCT *jct, PALLET *pall);

public:

 Central_Layer * add_C(Central_Layer *cl){
       cl->N = this; cl->networked = networked;
       return CLL.insert(cl);}                                          1980

 Network(int i):networked(i){}
 ~Network(){cerr<<"deleting Network\n";}
};

//*****************************************************************************
//******
```

```
//****** FUNCTIONS WHICH MUST BE DEFINDED BY SIMULATION PROGRAMMER
//******
                                                                        1990

//****** FROM WORKSTATION

// PALLET * WST::discipline()   WORKSTATION BUFFER DISCIPLINE

// int WST::pole(PALLET *)      used by LAYER_CONTROLER: returned value reflects
//                              how long it would take the WORKSTATION to service
//                              the referenced PALLET if submitted at time of
//                              polling
                                                                        2000

//****** FROM OUTput

// int OUT::pole(const IN_ORDER *)  returns some measure of how buissy the OUT
//                                  COMPonent is.  MY preferance is returning
//                                  the total number of PALLETs assigned to OUT

//****** FROM Input_Layer

                                                                        2010
// Next     Input_Layer::discipline(IN * in)
// int      Input_Layer::discharge(IN * in)

//****** FROM Central_Layer

// float Central_Layer::cost(int loc, PATH * path, PALLET * pall, WST* next_wst)

//****** FROM Output_Layer

// OUT * Output_Layer::destination(IN_ORDER *)  used by main to assign an ORDER an   2020
//                                              OUTput destination

//****** FROM Network

// virtual W_PATH Network::centralized_1(Central_Layer *CL, JCT * J, PALLET * P)
```

```
//*****************************************************************************

PALLET * WST::discipline(){  //   first in first out
    return b->remove((b->occu)-1);}                                          2030


int WST::pole(PALLET * P){   //   returns time (sec.) it will take WST to finish
    int count = 0;          //   processing the pallets in its buffer and machine
    if(b){           //   if buffer has a workstation; BUFFER's contribution
        for(int i=b->occu-1; i>=0; i--){
          count += p_time(b->buffer[i]->part_num());}}
    if(m->station){   //   if a PALLET is in WST machine; MACHINE's contribution
        count +=m->timer;}
    count +=p_time(P->part_num()); //   time WST MACHINE needs to process P   2040
    return count;}


int OUT::pole(const IN_ORDER *){  //   return number of pallets assigned to OUT
    return p_num;}


Next Input_Layer::discipline(IN * in, IN_QUEUE & q){
    Next n={0,0};
    IN_ORDER * r ;                                                           2050
    q.reset_oldest();            //   search from oldest to newest IN_ORDER
    while( (r = q.prev()) != 0){
      for(int k=0; k<PARTS; k++){ //   load IN with a product an IN_ORDER needs
        if((r->r_comp(k) >0) && in->part(k)){  //** part k found
          n.part = k; n.in_order = r;
          return n;}}}
    return n;}                   //   no product found in any IN_ORDER for IN


int  Input_Layer::discharge(IN * in){ //   no conditons, PALLETS go a.s.a.p.   2060
    return 1;}


float  Central_Layer::cost(PATH * path, PALLET * pall, WST* next_wst){
```

```
    //   RULE: only consider pallets in one layer when makeing decision

    int a=0; int size = path->size;
    for(int i=1; i<=(size-1); i++) a += path->list[i]->length();
    LNK * q = path->list[size-1];                                          2070
    float m_bar   = next_wst->m_time();
    int   to_proc = q->occu();
    float  b    = m_bar*to_proc;
    float  cost = (next_wst->pole(pall)) + a + b;
    return cost;}


OUT * Output_Layer::destination(IN_ORDER * in_o){//   OUT with lowest pole gets ORDER
    outs.reset();
    OUT * lucky = outs.next();                                             2080
    int score = lucky->pole(in_o);
    while(OUT * q = outs.next()){
        int a = q->pole(in_o);
        if(a < score){
            score = a; lucky = q;}}
    lucky->insert(*in_o);        //   create OUT_ORDER in lucky OUT
    in_o->out_stream(lucky);     //   assign lucky out to IN_ORDER
    return lucky;}

W_PATH Network::centralized_1(Central_Layer *cent_lay, JCT *jct, PALLET *pall){   2090

        float NEXT; Central_Layer *curr; WST *NEXT_WST; WST *q; PATH *p;
        int a = pall->part_num(); LNK * b = pall->last_tick_on();
        WL.clear();
        while( (curr = CLL.isIn(b->C_type().layer)) != 0){
        curr->wst_table()->reset();
        do{ //*** Find First Acceptable WST
           q = curr->wst_table()->next(); if(q==0) cerr<<"PALLET at dead end\n";
           p = curr->transit(b,q);} while( !(q->part(a) == 1 && p->size != 0));
                                                                           2100
           NEXT = curr->cost(p,pall,q);
           NEXT_WST = q;

        while( q = curr->wst_table()->next()){ //*** Find Best WST
```

```
    p = curr->transit(b,q);
    if(q->part(a) == 1 && p->size != 0){
      float n = curr->cost(p,pall,q);
      if(n < NEXT){
        NEXT = n; NEXT_WST = q;}}}
  WL.insert(NEXT_WST);                                          2110
  b= (LNK *) NEXT_WST->downstream();}
  W_PATH tmp(WL);
  return tmp;}
```

B.3 Testing Functions (SECtesting.h)

```
// Upstream & Downstream Testing Functions

  char *w[5]={"In","Link","Junction","Workstation","Out"};

  void down(COMP & ref){
    typ t1= ref.C_type();
    COMP * cta = ref.downstream();
    COMP * ctb = cta;
    if(cta){
      do{                                                              10
        typ t2= cta->C_type();
        cout  <<"Object downstream of "
              << w[t1.object] << "# "<<t1.id_num<<" in layer "<<t1.layer
       <<" is "<< w[t2.object] << "# "<<t2.id_num<<" in layer "<<t2.layer<<"\n";
        cta = ref.downstream();} while (cta != ctb);cout << "\n";}
    else{cout <<"No object downstream of "
              << w[t1.object] << "# "<< t1.id_num<<"\n\n";}}


  void up(COMP & ref){                                                 20
    typ t1= ref.C_type();
    COMP * cta = ref.upstream();
    COMP * ctb = cta;
    if(cta){
      do{
        typ t2= cta->C_type();
        cout  <<"Object upstream of "
              << w[t1.object] << "# "<< t1.id_num<<" in layer "<<t1.layer
       <<" is "<< w[t2.object] << "# "<< t2.id_num<<" in layer "<<t2.layer<<"\n";
        cta = ref.upstream();} while (cta != ctb);cout << "\n";}      30
    else{cout <<"No object upstream of "
              << w[t1.object] << "# "<< t1.id_num<<"\n\n";}}

// Auto-Configuration Testing Functions

 void give_dir(Central_Layer & cl,LNK * lnk, WST * wst){
   cout <<"From LINK #  "<< lnk->C_type().id_num <<" to WORKSTATION #  "
```

```
    << wst->C_type().id_num <<" in layer "<< wst->C_type().layer <<":\n";
  PATH tmp = *cl.transit(lnk,wst);
  if(tmp.size == 0){                                                    40
    cout<<"no connection\n\n"; return;}
  for(int i=0; i<tmp.size;i++){
    cout<<"LINK # "<<tmp.list[i]->C_type().id_num<<"\tin layer "
      << tmp.list[i]->C_type().layer <<"\n";}
  cout <<"\n\n";}

void  give_dir(Output_Layer & ol,LNK * lnk, OUT * out){
  cout <<"From LINK #  "<< lnk->C_type().id_num <<" to OUT #  "
    << out->C_type().id_num <<" in layer "<< out->C_type().layer <<":\n";
  PATH tmp = *ol.transit(lnk,out);                                      50
  if(tmp.size ==0){
    cout<<"no connection\n\n"; return;}
  for(int i=0; i<tmp.size;i++){
    cout<<"LINK # "<<tmp.list[i]->C_type().id_num<<"\tin layer "
      << tmp.list[i]->C_type().layer <<"\n";}
  cout <<"\n\n";}

//  List Testing Functions

void in_table_blah(Input_Layer & l){                                    60
  IN_table * table = l.in_table();
  table->reset();
  IN * q;
  cout<< "IN LIST size:     "<< table->size() <<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer<<"\n";}
  cout <<"\n\n";}

void lnk_table_blah(Central_Layer & l){                                 70
  LNK_table * table = l.lnk_table();
  table->reset();
  LNK * q;
  cout<< "LINK LIST size:     "<< table->size() <<"  Input_dim:     "<< l.in_dim()<<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();
```

```
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer;
  if(q->index != -1) cout << " Index # " << q->index <<"\n";
  else            cout << "\n";}
  cout <<"\n\n";}                                                          80

void jct_table_blah(Central_Layer & l){
  JCT_table * table = l.jct_table();
  table->reset();
  JCT * q;
  cout<< "JUNCTION LIST size:     "<< table->size() <<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer<<"\n";}
  cout <<"\n\n";}                                                          90

void wst_table_blah(Central_Layer & l){
  WST_table * table = l.wst_table();
  table->reset();
  WST * q;
  cout<< "W STATION LIST size:     "<<table->size()<<"  dim:     "<<l.out_dim()<<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer<<"\n";}
  cout <<"\n\n";}                                                          100

 void lnk_table_blah(Output_Layer & l){
  LNK_table * table = l.lnk_table();
  table->reset();
  LNK * q;
  cout<< "LINK LIST size:     "<< table->size() <<"  Input_dim:     "<<l.in_dim()<<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer;
  if(q->index != -1) cout << " Index # " << q->index <<"\n";               110
  else            cout << "\n";}
  cout <<"\n\n";}

 void jct_table_blah(Output_Layer & l){
  JCT_table * table = l.jct_table();
```

```
  table->reset();
  JCT * q;
  cout<< "JUNCTION LIST size:     "<< table->size() <<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();                                                    120
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer<<"\n";}
  cout <<"\n\n";}

 void out_table_blah(Output_Layer & l){
  OUT_table * table = l.out_table();
  table->reset();
  OUT * q;
  cout<< "OUT LIST size:     "<<table->size()<<"  dim:     "<<l.out_dim()<<"\n";
  while( (q=table->next()) != 0){
  typ tmp = q->C_type();                                                    130
  cout << w[tmp.object] <<" # "<<tmp.id_num<<" in layer "<<tmp.layer<<"\n";}
  cout <<"\n\n";}




//   print_order is used in test_sys.cc for run-time order arrival display

void print_order(IN_QUEUE & q, int arrived){
                                                                            140
cout << "************************************************************\n"
    << "**** Newly arrived orders in IN_QUEUE:\n"
    << "\n\n";
  IN_ORDER * r;
  q.reset_newest();  //****
  for(int i= 1; i<arrived;i++) q.next();
  while( (r = q.prev()) != 0){
      cout <<"ORDER:  #"<< r->id()
          <<"\tISSUED:  "
          <<r->ck_issue().day<<"/"<<r->ck_issue().hour<<":"                 150
          <<r->ck_issue().minute<<'\n'
          <<"\t\tDUE:  "
          <<r->ck_due().day  <<"/"<<r->ck_due().hour<<":"
          <<r->ck_due().minute<<'\n'
```

```
    <<"\t\tORDER SIZE\t"<<r->r_size()<<'\n'
    <<"\t\tORDER COMPOSITION\n\t\t";
for(int k=0; k <PARTS; k++){
    cout << r->r_comp(k) <<" ";}
cout <<"\t\tDESTINATION:  OUT#"<<r->out_stream()->C_type().id_num
    << "\n\n";}}                                                    160
```

Appendix C

Main Program - C++ (test_sys.cc)

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include "SECmyhead.h"
#include "SECLAYERS.h"
#include "SECtesting.h"

//*****************************************************************************
//***** LAYERS of System (must be at least 3)
                                                                          10
#define  LAYERS 4

//*****************************************************************************

Time t    ;
IN_QUEUE Q;
int fpoisson(float, long *);
struct ORDER forder_comp(int, long *);

int DE = 1000000;                                                         20

tasks T1_1={{1,0,1,1,0,1,1,0,1},{20,DE,15,20,DE,15,20,DE,15}};
tasks T1_2={{1,1,0,1,1,0,1,1,0},{40,45,DE,40,45,DE,40,45,DE}};
tasks T1_3={{0,1,1,0,1,1,0,1,1},{DE,25,35,DE,25,35,DE,25,35}};
tasks T2_1={{1,1,1,0,0,0,1,1,1},{15,15,15,DE,DE,DE,20,20,20}};
tasks T2_2={{1,1,1,1,1,1,0,0,0},{35,35,35,45,45,45,DE,DE,DE}};
```

```
tasks T2_3={{0,0,0,1,1,1,1,1,1},{DE,DE,DE,25,25,25,40,40,40}};

int prod_1[PARTS]={1,1,1,1,1,1,1,1,1};
int prod_2[PARTS]={1,1,1,1,1,1,1,1,1};
int prod_3[PARTS]={1,1,1,1,1,1,1,1,1};

main(){

//*****************************************************************************************
//******
//******      MPMS SYSTEM LAYER DECLARATIONS
//******

//********** LAYER O declarations

IN * in1 = new IN(0,1,prod_1); IN * in2 = new IN(0,2,prod_2);
IN * in3 = new IN(0,3,prod_3);

Input_Layer layer0 = *(new Input_Layer(0));

layer0.add_in(in1); layer0.add_in(in2); layer0.add_in(in3);

//********** LAYER 1 declarations

LNK * ga1 = new LNK(1,1,8); LNK * ga2 = new LNK(1,2,5); LNK * ga3 = new LNK(1,3,12);
LNK * ga4 = new LNK(1,4,4); LNK * ga5 = new LNK(1,5,6); LNK * ga6 = new LNK(1,6,7);
LNK * ha7 = new LNK(1,7,7); LNK * ha8 = new LNK(1,8,6); LNK * ha9 = new LNK(1,9,10);
LNK * ha10= new LNK(1,10,9);

JCT * ja1 = new JCT(1,1); JCT * ja2 = new JCT(1,2); JCT * ja3 = new JCT(1,3);

WST * wsta1 = new WST(1,1,5,T1_1); WST *  wsta2 = new WST(1,2,10,T1_2);
WST * wsta3 = new WST(1,3,20,T1_3);

Central_Layer layer1 = *(new Central_Layer(1));

layer1.add_lnk(ga1); layer1.add_lnk(ga2); layer1.add_lnk(ga3);
layer1.add_lnk(ga4); layer1.add_lnk(ga5); layer1.add_lnk(ga6);
layer1.add_lnk(ha7); layer1.add_lnk(ha8); layer1.add_lnk(ha9);
```

```
layer1.add_lnk(ha10);

layer1.add_jct(ja1); layer1.add_jct(ja2); layer1.add_jct(ja3);

layer1.add_wst(wsta1); layer1.add_wst(wsta2); layer1.add_wst(wsta3);                    70

//********** LAYER 2 declarations

LNK * gb1 = new LNK(2,1,9) ; LNK * gb2 = new LNK(2,2,5); LNK * gb3 = new LNK(2,3,8);
LNK * gb4 = new LNK(2,4,3) ; LNK * gb5 = new LNK(2,5,4); LNK * gb6 = new LNK(2,6,6);
LNK * hb7 = new LNK(2,7,15); LNK * hb8 = new LNK(2,8,9); LNK * hb9 = new LNK(2,9,14);
LNK * hb10= new LNK(2,10,6);

JCT * jb1 = new JCT(2,1); JCT * jb2 = new JCT(2,2); JCT * jb3 = new JCT(2,3);
                                                                                         80
WST * wstb1 = new WST(2,1,10,T2_1); WST *  wstb2 = new WST(2,2,20,T2_2);
WST * wstb3 = new WST(2,3,30,T2_3);

Central_Layer layer2 = *(new Central_Layer(2));

layer2.add_lnk(gb1); layer2.add_lnk(gb2); layer2.add_lnk(gb3);
layer2.add_lnk(gb4); layer2.add_lnk(gb5); layer2.add_lnk(gb6);
layer2.add_lnk(hb7); layer2.add_lnk(hb8); layer2.add_lnk(hb9);
layer2.add_lnk(hb10);
                                                                                         90
layer2.add_jct(jb1); layer2.add_jct(jb2); layer2.add_jct(jb3);

layer2.add_wst(wstb1); layer2.add_wst(wstb2); layer2.add_wst(wstb3);

//********** LAYER 3 declarations

LNK * gc1 = new LNK(3,1,8) ; LNK * gc2 = new LNK(3,2,9); LNK * gc3 = new LNK(3,3,12);
LNK * gc4 = new LNK(3,4,4) ; LNK * gc5 = new LNK(3,5,7); LNK * gc6 = new LNK(3,6,3);
LNK * hc7 = new LNK(3,7,10); LNK * hc8 = new LNK(3,8,7); LNK * hc9 = new LNK(3,9,9);
LNK * hc10= new LNK(3,10,8);                                                            100

JCT * jc1 = new JCT(3,1); JCT * jc2 = new JCT(3,2); JCT * jc3 = new JCT(3,3);

OUT * out1 = new OUT(3,1); OUT *  out2 = new OUT(3,2); OUT * out3 = new OUT(3,3);
```

```
Output_Layer layer3 = *(new Output_Layer(3,"LF_333"));

layer3.add_lnk(gc1); layer3.add_lnk(gc2); layer3.add_lnk(gc3);
layer3.add_lnk(gc4); layer3.add_lnk(gc5); layer3.add_lnk(gc6);                    110
layer3.add_lnk(hc7); layer3.add_lnk(hc8); layer3.add_lnk(hc9);
layer3.add_lnk(hc10);

layer3.add_jct(jc1); layer3.add_jct(jc2); layer3.add_jct(jc3);

layer3.add_out(out1); layer3.add_out(out2); layer3.add_out(out3);

//*****************************************************************************
//******
//******                    Network decleration                                 120
//******

Network NET= *(new Network(0)); //**** 1 => route based scheduling policy
                        //**** 0 => local feedback scheduling policy

NET.add_C(&layer1);
NET.add_C(&layer2);

//*****************************************************************************
//******                                                                        130
//******                    SYSTEM OBJECT linking
//******

  join(*in1,*ga1) ; join(*in2,*ga2)  ; join(*in3,*ga3)  ;  join(*ga1,*ja1)  ;
  join(*ga2,*ja2) ; join(*ga3,*ja3)  ; join(*ja1,*ha7)  ;  join(*ha7,*ja2)  ;
  join(*ja2,*ha9) ; join(*ha9,*ja3)  ; join(*ja3,*ha10) ;  join(*ha10,*ja2) ;
  join(*ja2,*ha8) ; join(*ha8,*ja1)  ; join(*ja1,*ga4)  ;  join(*ja2,*ga5)  ;
  join(*ja3,*ga6) ; join(*ga4,*wsta1); join(*ga5,*wsta2);  join(*ga6,*wsta3);

                                                                              140
  join(*wsta1,*gb1); join(*wsta2,*gb2) ; join(*wsta3,*gb3) ;  join(*gb1,*jb1)  ;
  join(*gb2,*jb2)  ; join(*gb3,*jb3)   ; join(*jb1,*hb7)   ;  join(*hb7,*jb2)  ;
  join(*jb2,*hb9)  ; join(*hb9,*jb3)   ; join(*jb3,*hb10)  ;  join(*hb10,*jb2) ;
```

```
  join(*jb2,*hb8)  ; join(*hb8,*jb1)   ; join(*jb1,*gb4)   ;  join(*jb2,*gb5)  ;
  join(*jb3,*gb6)  ; join(*gb4,*wstb1) ; join(*gb5,*wstb2) ;  join(*gb6,*wstb3);

  join(*wstb1,*gc1); join(*wstb2,*gc2) ; join(*wstb3,*gc3) ;  join(*gc1,*jc1)  ;
  join(*gc2,*jc2)  ; join(*gc3,*jc3)   ; join(*jc1,*hc7)   ;  join(*hc7,*jc2)  ;
  join(*jc2,*hc9)  ; join(*hc9,*jc3)   ; join(*jc3,*hc10)  ;  join(*hc10,*jc2) ;
  join(*jc2,*hc8)  ; join(*hc8,*jc1)   ; join(*jc1,*gc4)   ;  join(*jc2,*gc5)  ;                    150
  join(*jc3,*gc6)  ; join(*gc4,*out1)  ; join(*gc5,*out2)  ;  join(*gc6,*out3) ;

//*****************************************************************************************
//******
//******                    System Auto-Configuration
//******

layer1.make_transit();
layer2.make_transit();
layer3.make_transit();                                                                          160

//*****************************************************************************************
//******
//******     ORDER_Q
//******

int fpoisson(float, long *);
struct ORDER forder_comp(int, long *);

 long idum =  -21415926 ;                                                                       170
 ORDER s              ;

  for(int k = 0; k<= 5*345600; t.inc(), k++){ //   simulate for about 480 hours
   if( (t.ck().hour%1    == 0) &&          //   order arivals every 15 minutes
       (t.ck().minute%1  == 0) &&
       (t.ck().second%60 == 0) &&
       (k<=5*345600)){                  //   orders arive for first 480 hours only
      int arrivals = fpoisson(0.0333,&idum);//**** mean interarival time= 30 minutes

      for(int j=1; j<=arrivals; j++){                                                           180
       s=forder_comp(100,&idum);  //**** average order size 100/2=50
       Q.insert(s);               //**** put order in ORDER_Q
```

```
        Q.reset_newest();
        IN_ORDER * io_ptr = Q.next();
        layer3.destination(io_ptr);}
           if(arrivals) print_order(Q,arrivals);}

    layer3.dt();
    layer2.dt();
    layer1.dt();                                                    190
    layer0.dt();}
return 0;}
```

200

Bibliography

[1] A. Bauer, R. Bowden, et al. *Shop Floor Control Systems: From Design to Implementation*, pages 3–12. Chapman & Hall, London, 1994.

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Co., Reading MA, 1995.

[3] O. Patrick Kreidl. Distributed cooperative control architectures for automated manufacturing systems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, February 1996.

[4] Grace Y. Lin and James J. Solberg. *Computer Control of Flexible Manufacturing Systems: Research and Development*, pages 169–206. Chapman & Hall, London, 1994.

[5] David M. Papurt. *Inside the Object Model: The Sensible Use of C++.* SIGS Books, New York, 1995.

[6] William H. Press, Saul A. Tevkolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*, pages 287–295. Cambridge University Press, New York, 1992.

[7] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[8] K. Sam Shanmugan. *Random Signals: Detection, Estimation, and Data Analysis.* John Wiley & Sons, Inc., New York, 1988.